2019

ACCELERATING DATA ACCESSING BY EXPLOITING FLASH MEMORY TECHNOLOGIES

Jing Yang
University of Rhode Island, jingyangcn@gmail.com

ACCELERATING DATA ACCESSING BY EXPLOITING FLASH MEMORY TECHNOLOGIES

BY

JING YANG

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

ELECTRICAL, COMPUTER AND BIOMEDICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2019

DOCTOR OF PHILOSOPHY DISSERTATION

OF

JING YANG

APPROVED:

Dissertation Committee:

Major Professor Qing Yang

Resit Sendag

Ying Zhang

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2019

ABSTRACT

Flash memory based SSD (solid state Drive) receives a lot of attention recently. SSD is a semiconductor device which provides great advantages in terms of high-speed random reads, low power consumption, compact size, and shock resistance. Traditional storage systems and algorithms are designed for hard disk drive (HDD), they do not work well on SSD because of SSD's asymmetric read/write performances and unavoidable internal activities, such as garbage collection (GC). There is a great need to optimize current storage systems and algorithms to accelerate data access in SSD. This dissertation presents four methods to improve the performance of the storage system by exploiting the characteristics of SSD.

GC is one of the critical overhead of any flash memory based SSD. GC slows down I/O performance and decreases endurance of SSD. This dissertation introduces two methods to minimize the negative impact of GC, "WARCIP: Write Amplification Reduction by Clustering I/O Pages" and "Thermo-GC: Reducing Write Amplification by Tagging Migrated Pages during Garbage Collection". WARCIP uses a clustering algorithm to minimize the rewrite interval variance of pages in a flash block. As a result, pages in a flash block tend to have a similar lifetime, minimizing valid page migrations during GC. The idea of Thermo-GC is to identify data's hotness during GC operations and group data that have similar lifetimes to the same block. Thermo-GC can minimize valid page movements and reduce GC cost through clustering valid pages based on their hotness. Experiment results show that both WARCIP and Thermo-GC can improve the performance of SSD and reduce data movements during GC, implying extended lifetimes of SSDs.

SSD fits naturally as a cache between the system RAM and the hard disk drive due to its performance/cost characteristics. But traditional cache replacements are designed for the hard disk drive, which do not work well on SSD because of SSD's

asymmetric read/write performances and wearing issues. In this dissertation we present a new cache management algorithm. The idea is not to cache data in SSD upon the first access. Instead, SSD caches when data are determined to be hot enough and warrant caching in the SSD. Data cached in the SSD is managed using an asymmetrical replacement policy for read/write by means of conservative promotion upon hits.

The nonvolatile characteristic of SSD allows cached data persistent even after power failures or system crashes. So the system can benefit from a hot restart. Current researches on SSD caches mainly focused on cache architecture or management algorithms to optimize performance under normal working conditions. Few studies were done on exploiting SSD caches durability across system crashes or power failures. To enhance storage system performance with a hot restart, a new metadata update technique is proposed in this dissertation to maximize usable data upon a system restart. The new metadata update method runs two-fold faster than Facebook's Flashcache after recovering from a system crash or power failure.

Overall, this dissertation describes two methods that reduce valid page migrations caused by SSD's internal activities, and two cache management algorithms that improve the performance of SSD cache. These four methods overcome current shortcomings in SSD's usage in industrial and improve storage system performance efficiently.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my major professor Qing Yang for his guidance, encouragement and support during my Ph.D. study. Dr. Qing Yang introduced me to the University of Rhode Island and trained me to be a researcher through teaching me scientific writing and presenting skills. I am grateful for his professional mentorship and personal advice.

I would like to thank Dr. Resit Sendag and Dr. Haibo He for giving insightful suggestions in my research projects. I would like to thank Dr. Ying Zhang and Dr. Noah Daniels for serving in my dissertation committee and commenting on my dissertation. I would also like to extend my thanks to faculty and staff members of the department of Electrical, Computer, and Biomedical Engineering, in particularly to Dr. Bin Li, Dr. Jien-chung Lo, Dr. Kunal Mankodiya, Dr. Yan Sun, Dr. Tao Wei, Dr. Frederick j. Vetter, Ms. Lisa Pratt and Ms. Meredith Leach for their help throughout my Ph.D. study at the University of Rhode Island.

I would also like to thank Shuyi Pei for her valuable suggestions in trouble-shooting and help in building prototypes. I am thankful for her encouragement and collaborative attitude when the project is not going smoothly. I appreciate Dongyang Li's support in my Ph.D. study and introduces me to hardware developing.

I owe my deepest thankfulness to my parents, they are my strongest supporters and offer unconditional love. I wish to give my sincere thanks and appreciation to my wife Zhengxi Wei for her love and unfailing support. She fills my life with colors. I cherish the time we spend together, no matter how far we are. I am lucky to have a thoughtful partner like her. I am looking forward to the coming of our first baby and starting a new chapter of our life together.

PREFACE

This dissertation is prepared according to the manuscript format, consisting of four manuscripts. A brief information of the manuscripts are as follows.

Manuscript 1: "WARCIP: write amplification reduction by clustering I/O pages" is published on the 12th ACM International Conference on Systems and Storage.

Manuscript 2: "Thermo-GC: Reducing Write Amplification by Tagging Migrated Pages during Garbage Collection" is accepted by the 14th IEEE International Conference on Networking, Architecture and Storage.

Manuscript 3: "F/M-CIP: Implementing Flash Memory Cache Using Conservative Insertion and Promotion" is published on the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.

Manuscript 4: "A New Metadata Update Method for Fast Recovery of SSD Cache" is published on the 8th IEEE International Conference on Networking, Architecture and Storage.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

MANUSCRIPT 1

WARCIP: Write Amplification Reduction by Clustering I/O Pages

by

Jing Yang[1], Shuyi Pei[2], Qing Yang[3]

is published on the 12th ACM International Conference on Systems and Storage.

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: jyang@ele.uri.edu

[2]Graduate Student, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: spei@ele.uri.edu

[3]Distinguish Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: qyang@ele.uri.edu

Abstract

The storage volume of SSDs has been greatly increased recently with emerging multi-layer 3D triple-level cell and quad-level cell. However, one critical overhead of any flash memory SSD is the garbage collection (GC) process that is necessary due to the inherent physical property of flash memories. GC is a time consuming process that slows down I/O performance and decreases endurance of SSD. To minimize the negative impact of GC, we introduce Write Amplification Reduction by Clustering I/O Pages (WARCIP). The idea is to use a clustering algorithm to minimize the rewrite interval variance of pages in a flash block. As a result, pages in a flash block tend to have a similar lifetime, minimizing write amplification during a garbage collection. We have implemented WARCIP on an enterprise NVMe SSD. Both simulation and measurement experiments have been carried out. Real world I/O traces and standard I/O benchmarks are used in our experiments to assess the potential benefit of WARCIP. Experiment results show that WARCIP reduces write amplification dramatically and the number of block erasures by 4.45 times on average, implying extended lifetimes of flash SSDs.

1.1 Introduction

Flash memory solid-state drives (SSDs) have fundamentally changed the landscape of data storage market and continue its broader applications due to its increased performance and reduced cost. With the emergence of 3D triple-level cell (TLC) and quad-level cell (QLC) flash technologies, modern SSDs can offer storage capacity of tens of terabytes (TB). In addition to huge volume, SSD's high throughput meets the requirements for enterprise applications by virtue of its massively parallel architecture at the back-end including multiple channels, multiple dies per channel, multiple logic units (LUNs) per die, and multiple planes per LUN. Parallel I/O operations can be performed among independent planes giving rise to a great potential for very high I/O performance.

One major concern of SSD's deployment, however, is the performance degradation and variation caused by its unavoidable internal activities, garbage collection (GC). GC consumes a significant amount of back-end bandwidth and increases the latency of regular I/O requests by a factor of 100 [1]. The key reason why GC drags down I/O performance is Write Amplification (WA), because valid pages have to be migrated from a block to be garbage collected (erased). WA not only slows down I/O performance but also decreases the endurance of SSD. Hence, there is a pressing need for minimizing WA and reducing GC overheads.

Extensive research has been reported in the literature on mitigating the impact of GC [2, 3, 4, 5, 6, 7]. In traditional SSDs that have only one appending point (open block) for SSD writes, data pages with various lifetimes are very likely mixed in the same erase unit [8, 9]. Multi-Stream technology [8] emerged recently has multiple open blocks in an SSD and allows data with different stream IDs to be stored in different erase units. Leveraging the Multi-Stream technology, many techniques have been proposed [8, 9, 10, 11]. Multi-Stream SSD [8] exposes

its streams to applications and let applications decide which stream to be used. FStream [10] automatically maps file system generated data to different streams. PCStream [11] automates the stream selection based on program counters. AutoStream [9] provides an automated streams classification strategy based on data updating frequency and recency to assign data to different streams.

While existing researches have made advances in terms of minimizing WA and GC cost, rapid changing I/O workloads in ever widening applications still impose many challenges to enterprise SSD designs. First of all, due to stream resource limitations, data with different lifetimes may be assigned to the same stream, especially in the cloud environment. While Multi-Stream SSD only supports 8 to 16 streams, dozens of processes may write simultaneously [12, 13, 14, 11]. There is a high chance that two processes with different data update intervals share the same stream ID. Furthermore, all existing automatic stream management strategies classify writing data into a limited number of streams with fixed classification rules. Each stream covers a relatively large range of data's lifetimes and cannot be quickly self-optimized for different workloads. In this situation, when workload changes, some streams can be extremely busy while others are idle wasting the stream resources.

To tackle these challenges and make SSD a viable solution for enterprises with steadily sustainable I/O performance and satisfactory endurance, we introduce Write Amplification Reduction by Clustering I/O Pages (WARCIP). The principal idea of WARCIP is minimizing rewrite interval (RWI) variance of pages in a flash block. RWI is the time interval of two consecutive write requests for the same logical block address. WARCIP calculates RWI of each write request in the I/O stream and groups it into an open block whose average RWI is closest. What is more, WARCIP learns current I/O patterns to fine-tune its clustering activities

adaptively. It merges, splits, and adjusts clusters automatically to achieve the best clustering result. While WARCIP only has several clusters (open blocks) at a time point, it has no upper limit in terms of how many clusters it can support. WARCIP is able to quickly change its grouping strategy to adapt to new workload characteristics by closing several open blocks and open new ones. Therefore, WARCIP can group pages with similar RWIs to the same physical block efficiently and accurately.

In order to assess how efficient the newly proposed WARCIP is in terms of I/O performance and endurance, We have carried out extensive experiments on both simulator and real SSD. Our simulator shares the same FTL design that is used in our in-house enterprise-level SSD prototype. The simulator uses page-level address mapping. Whenever an open block is filled, the simulator chooses the block that has the lowest erasure count as the new open block. When the number of free blocks is lower than a threshold, the block that has the least valid pages is selected by GC. To support Multi-Stream, 8 open blocks are maintained. After extensive simulation experiments, we have implemented a fully functional WARCIP on our NVMe SSD prototype. The SSD prototype aims at enterprise applications using Marvell's "Zao" NVMe SSD controller and Micron TLC NAND flash memory chips with total storage capacity of 8TB. The fully functioning prototype SSD has PCIe interface at the front end and it has 4GB on-board DDR4 SDRAM. At the back-end, it has 8 channels, 4 dies per channel, 2 luns per die and 4 planes per lun. The SSD is plugged into a PCIe Gen3x8 slot of a storage server that generates I/O activities to the SSD based on standard I/O benchmarks and real-world I/O traces. Experimental results show that write amplification is reduced dramatically and the number of block erasures drops to as low as one-tenth of the traditional SSDs. This paper makes the following contributions.

- Our analyses and measurements have discovered that clustering pages in open blocks is more effective than classifying data streams in the host. Clustering pages into an open block directly reduces rewrite interval variance of pages in the block, minimizing WA.

- A novel self-optimizing and adaptive clustering algorithm is proposed, referred to as Write Amplification Reduction by Clustering I/O Pages (WARCIP). WARCIP can minimize rewrite interval variance in a block by clustering write requests according to their rewrite intervals. It is adaptive to and self-optimized for different workloads through a reward mechanism.

- A fully functioning WARCIP has been implemented in an enterprise PCIe SSD card. Comprehensive experimental evaluations have been carried out to show WARCIP's advantages over SSDs using the state-of-the-art WA minimization techniques.

The rest of this paper is structured as follows. WARCIP's design and implementation will be given in Sections 1.2 and 1.3, respectively. Section 1.4 describes the experimental setup and workload characteristics. In Section 1.5, experiment results will be presented and discussed. Related works are discussed in Section 1.6. We conclude our work in Section 1.7.

1.2 WARCIP Design

WARCIP seeks to intelligently minimize lifetime variance of pages in a flash block. To efficiently and accurately achieve this, WARCIP takes the following three strategies. Firstly, RWI, the write request interval of the same logical block address, is introduced to quantitatively characterize pages' lifetime based on LBA access patterns (Section 1.2.2). Secondly, to meet the stringent time and resource constraints of enterprise SSDs, we propose an on-line self-optimized clustering

algorithm which can quickly tune its grouping strategy to adapt to changing I/O patterns (Section 1.2.3). Thirdly, a rewarding feedback mechanism is used to achieve global optimization (Section 1.2.4). The overall structure of WARCIP is shown in Figure 1.

1.2.1 SSD Internal Operations

Before presenting WARCIP, we first give a brief overview of internal operations of modern SSD that are essential for understanding WARCIP design.

Log-structured Programming: NAND flash memory cells can only be changed in one direction (e.g., from 1 to 0 or 11 to 10, but not vice versa). No in-place write (program) can be executed in NAND flash based SSD. Hence, each write request requires a new page (clean page) to store new data and turns the corresponding obsolete data from valid to invalid if any. Flash translation layer (FTL) is designed to manage this one-way write manner. It maintains a mapping from logical address to physical address and tracks the status of each page. To facilitate the work of FTL, SSD organizes new writes in a log-structured way, new data is always appended at the end of an open block.

Garbage Collection and Write Amplification: When the number of clean blocks reduces to a certain amount, GC is triggered to generate clean pages [15]. The block chosen for GC often contains valid pages. Before erase operation can be started on that block, valid pages have to be moved out. Valid pages migration is known as write amplification (WA) because of the extra writes of the same content. The more valid pages there are in an erasing block, the larger WA will be. WA consumes SSD back-end bandwidth and drags down its I/O performance significantly.

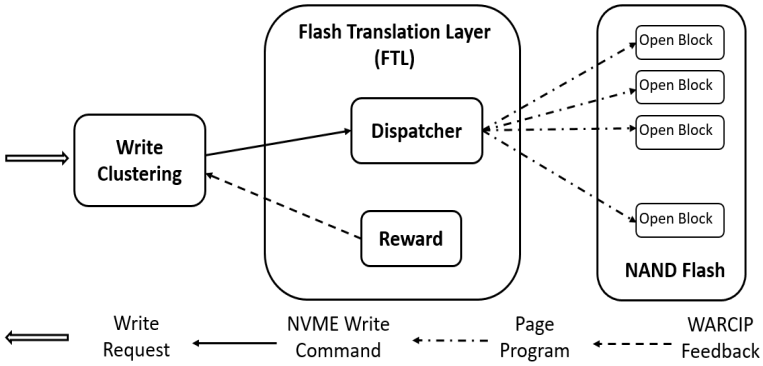


Figure 1: High-level overview of WARCIP.

1.2.2 Rewrite Interval Measurement

To properly predict the lifetime of a flash page for optimal clustering, intuition and experiments should be used to analyze typical I/O behaviors, especially in different environments. For example, users typically edit files and periodically save their edited files to persistent storage. When such operation is performed, a bunch of pages will be written to the storage at the same time with the same update cycle in adjacent locations. For a server equipped with buffer cache and page cache, cache management algorithms flush a bunch of pages across a large logical address range. In online transaction processing, many small transactions submitted at the same time generate mixed pages and store them in a random way. The same situation (pages mixed) happens in the virtual machine environments such as VMWare and Xen. In all above scenarios, LBAs of these write requests may or may not be in proximity address range. Even though write requests arrive at the same time, they have different update cycles. Therefore a fine-grained page-level rewrite interval measurement is necessary to calculate update cycles accurately.

It is ideal to measure the lifetime of data at page-level, which allows maintaining the lifetime of every page in SSD. However, tracking at page-level is very costly in terms of DRAM usage. Although DRAM price is decreasing and enterprise SSDs have multiple GBs of on board DRAM, such DRAM is still precious resources that

store FTL table and cache data. Analyzing 18 cloud server traces [16] from a production environment, we observed that average write request size of most servers is larger than 8KB. Therefore, we measure rewrite intervals on multiple pages to reduce DRAM consumption while maintaining sufficient accuracy. Specifically, WARCIP maintains a time stamp for 8 consecutive logical pages. These 8 consecutive pages form the basic measuring unit for our WARCIP implementation. A 4-byte time stamp is kept associated with each measuring unit and is refreshed whenever any of the 8 pages are updated. Based on this time stamp, WARCIP calculates RWI for each unit for the prediction of data life. When a write request arrives, its RWI is calculated by subtracting the recorded time stamp from current system time.

1.2.3 Clustering

Clustering data into groups is one of the most fundamental methods of understanding and learning. It has long been used in many scientific fields. The most popular and straightforward clustering algorithms are streaming k-means algorithms [17, 18]. They can cluster a stream into k groups in one-pass. Streaming k-means algorithms are light-weight and practical. They have been successfully used in many applications. However, direct application of streaming k-means algorithms in our situation would not work for the following reasons. First of all, clustering write requests in SSD requires that the size of a cluster is fixed and equals the size of a flash block, but streaming clustering algorithms do not restrict the size of a cluster. Secondly, the number of clusters is unknown in advance and depends on the number of flash blocks used. However, streaming clustering algorithms are either required to define cluster number in advance or have very high time complexity. Thirdly, the performance target of our SSD is 1 million IOPS since it is for enterprise applications. Such high IOPS require that the clustering

algorithm make a decision in less than one microsecond on an embedded system with computing and storage resource restrictions. Fourthly, since write pattern changes over time, clustering policy needs to be tuned and adjusted continuously to capture new IO patterns correctly.

To deal with these challenges, WARCIP adopts the following two strategies: Greedy Clustering and Dynamic Split-and-Merge as explained in the following paragraphs.

Greedy Clustering

Traditional clustering algorithms cluster data into a limited number of clusters regardless of the size of each cluster. However, in SSD, the size of each cluster should be equal to the size of an open block. In addition, traditional clustering algorithms find solutions iteratively, while SSD has to cluster write requests as soon as they arrive in one shot. In practice, clusters are only effective before the next GC. Dated write requests should not be clustered with current one even they have the same RWI. Therefore, we propose the following greedy clustering strategy.

Assuming there are k open blocks (clusters) in the system, WARCIP calculates the average RWI of each cluster, designated as the *center* of the cluster, u_k. An incoming write request is appended to an open block whose center is closest to the RWI of the write request. The center u_k is immediately updated to the new average RWI. When an open block is full, WARCIP chooses a free block as a new open block and closes the old one. The center of the new open block is initialized equidistantly from the nearest two centers. Although WARCIP only supports k clusters at a certain time point, it can potentially support an unlimited number of clusters by continually opening new clusters with different centers. Each cluster corresponds to one open block that contains multiple pages ranging from 64, 128, to 256 or more, depending on the type of flash memories (e.g., SLC, MLC, or

TLC).

Dynamic Split-and-Merge

Write requests arrive continuously with evolving patterns. Clustering on outdated information may not reflect IO pattern changes and result in poor clustering quality. For example, when I/O pattern changes from inactive to active, it is possible that RWIs of all bursty I/O requests are less than the minimum average RWI of all open blocks. If all bursty I/Os go into the same open block, the block becomes extremely busy, while other open blocks may be inactive or idle that wastes the limited open block resources. Moreover, situations become even worse when SSD is full. Our extensive experiments have revealed a very important phenomenon: Flash blocks that absorb bursty I/Os contributes to a noticeable portion of WA. This is due to the fact that, when SSD is short of free blocks, the block that absorbs bursty I/Os is likely to become erase candidate because it will contain the least number of valid pages in a short time. Although the number of valid pages is small during each GC, it still causes noticeable WA due to the high erasing frequency. In addition, such blocks increase GC activities significantly, which adversely affect front-end I/O performance. If we can separate the bursty I/O requests to more open blocks based on their rewrite interval times, we can further reduce WA.

For this purpose, we introduce a dynamic split-and-merge (D-SAM) mechanism. When WARCIP detects more than half of the write requests are assigned to an open block in a time window, it splits the open block into two by adding a new grouping center and adjusting the old one. As shown in Figure 2(a), a new open block is spun off for the new RWI range, and bursty I/Os are separated to two clusters accordingly. In this scenario, both clusters have lower rewrite interval variances, thus WA is reduced. On the contrary, when I/O pattern changes, some open blocks may become inactive containing pages partially filling the blocks. To

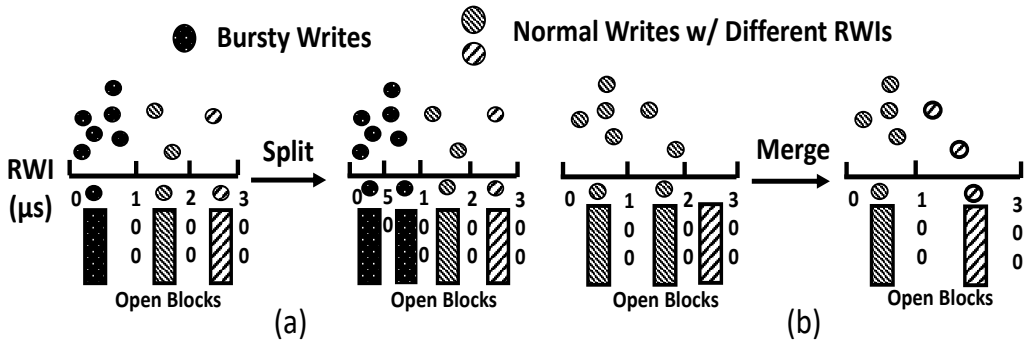


Figure 2: Examples of split (a) and merge (b) operations.

better utilize the limited number of open blocks, WARCIP merges such blocks and closes an inactive open block. The merge process is shown in Figure 2(b). When an open block has not received enough pages in a time window, WARCIP closes the cluster by merging it with its closest neighbor cluster as shown in Figure 2(b). D-SAM operation is triggered periodically in our WARCIP implementation. Let N be the number of open blocks that have been closed (written to flash) since the last D-SAM operation. Whenever N reaches a threshold T, D-SAM starts. T can be tuned depending on applications. Our experiments have shown that D-SAM performs very well by setting $T = 256$. That is, the Dynamic Split-and-Merge operation is activated whenever 256 blocks are filled or closed.

1.2.4 GC Feedback

To this point, WARCIP's clustering has been solely based on current rewrite interval information. Whether or not such clustering delivers the desired performance can be assessed by what happens at GC time. If the clustering is done well, then the number of valid pages in a GC block is small. Otherwise, we may have a large number of valid pages that have to be migrated, indicating a poor clustering. In order to give proper credits and blames to the past clustering actions so that the system can reach an optimal state, WARCIP is equipped with a feedback and rewarding mechanism, adding another layer of learning capability.

In a properly clustered block, all pages in that flash block should be rewritten in a short time window after one of them is updated. If this were the case, most pages in the block would have been invalidated when the block is to be erased. Such clustering should receive a positive reward. In contrast, if pages are not rewritten before the block is selected to be erased in a GC process, those valid pages need to be copied to other blocks before being erased. This implies those pages have been placed in a wrong cluster. In this case, WARCIP should blame the wrong action of clustering for these pages and give a negative reward to these pages because they have a longer RWI than other pages in the block.

To quantify the rewarding mechanism, positive or negative value should be assigned to the past actions on clustering. Considering the fact that FTL does not maintain the mapping table for invalid physical pages, it may incur extra overhead to provide positive rewards. Our strategy is "no news is good news", but bad actions should be reported. Reporting bad actions is relatively easy since wrongly grouped pages are still valid and their information is still available in FTL. To feedback negative reward to indicate wrong grouping, WARCIP doubles the RWI of these valid pages, and then they will be grouped with other pages that have larger update intervals in the future. Doubling RWI incurs very little overhead and achieve the purpose of giving negative rewards to the wrong clustering actions.

1.3 Implementation

In this section, we discuss implementation details of WARCIP by taking into account the practical factors of real systems.

1.3.1 WARCIP Architecture

The overall WARCIP architecture is shown in Figure 3. WARCIP consists of WARCIP driver, WARCIP dispatcher and WARCIP reward collector. The

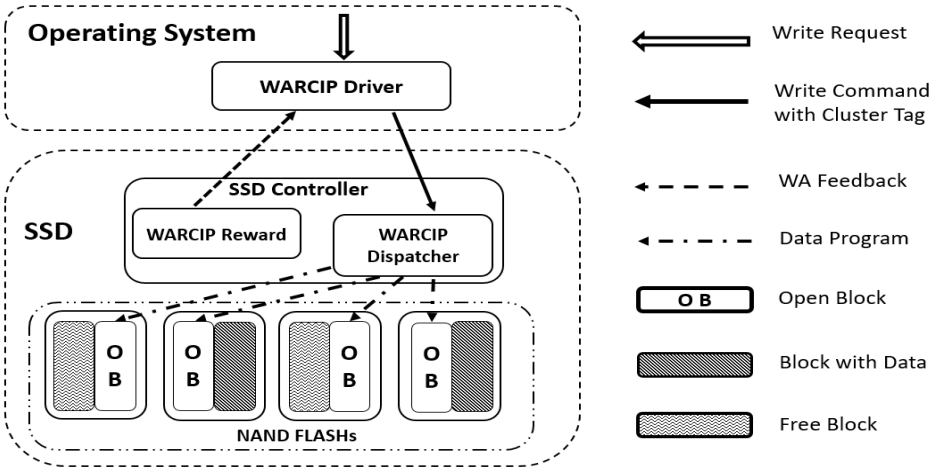


Figure 3: The system architecture of WARCIP.

WARCIP driver executes clustering algorithms as described in Section 1.2.3. It calculates the RWI of each write request, then assigns it to the open block with the nearest average RWI by tagging the open block number in the NVMe write command. WARCIP dispatcher guides FTL to dispatch write commands to the corresponding open blocks according to the attached block number. WARCIP reward collector module collects the information of WA, such as LBAs of pages that are migrated to other blocks due to garbage collection, and waits for polling by WARCIP driver. Clustering algorithms are implemented in the driver because our SSD prototype is still under development (pre-alpha version) and being tuned for better I/O performance. In alpha version, all algorithms will be embedded in FTL which makes SSD with WARCIP a plug and play device.

1.3.2 WARCIP Driver Initialization

When WARCIP driver is loaded into the operating system for the first time, it obtains the configuration (e.g., page size, number of pages in a block, number of open blocks) of SSD by sending a query to it. WARCIP driver uses this information to allocate the necessary resources and configure the learning logic.

The WARCIP driver maintains time stamps in a table to track rewrite inter-

vals. Each time stamp is a 4-byte variable that keeps the latest access time of a page. As mentioned in Section 1.2.2, each time stamp tracks 8 consecutive pages. If any of the 8 pages is updated, the corresponding time stamp are updated to the current time. The WARCIP driver also maintains the status of open blocks. The maximum number of open blocks is determined by the SSD. To keep time stamps persistent across reboots, all time stamps are stored in NAND flashes during system shutdown and restored to DRAM after reboot. When a power failure happens, the time stamp table is built from scratch.

1.3.3 Request Processing

After getting the open block number of current write request, WARCIP embeds it in DWORD 12 of the write command according to the NVM Express specification [19] and sends it to SSD. Inside SSD, WARCIP dispatcher decodes the open block number from the write command and passes it to FTL. FTL finds the corresponding open block and translates the write request's LBAs to physical page addresses of that block.

1.3.4 GC Feedback

WARCIP reward collector is implemented in SSD firmware. It collects WA information (LBAs of the pages that need to migrate) during each GC and sends them back to the WARCIP driver. WARCIP driver periodically takes these LBAs back to host. After getting LBAs of those pages that were migrated during a GC and caused WA, WARCIP driver punishes all migrated page's RWIs by recalculating the RWIs based on current system time. The RWIs of these pages are effectively doubled by subtracting the RWIs from the time stamp that is stored in the table.

1.4 Experimental Setup and Workload Characteristics

Experimental Settings: We have implemented a fully functional WARCIP on our enterprise level NVMe SSD prototype. The NVMe SSD prototype uses Marvell's "Zao" NVMe SSD controller and Micron TLC NAND Flashes with a total storage capacity of 8TB. The fully functioning prototype has 4GB on-board DDR4 SDRAM. At the back-end of the SSD, there are 8 channels, 4 dies per channel, 2 luns per die and 4 planes per lun. Each plane contains 504 flash blocks, each of which holds 2304 flash pages. It is plugged into a PCIe Gen3x8 slot of a Linux server with Intel i7-7700K CPU, 8GB DRAM, EXT4 filesystem and its Kernel version is 4.4. During experiments, the size of SSD is limited to 180GB with 18% over-provisioned (OP) space to make SSD quickly reaches its steady-state (i.e., all pages in SSD have been written at least once, and GC has been triggered).

Since WARCIP is time-dependent, our measurement experiment takes an entire week to complete run through the one week's I/O traces. In order to better evaluate WARCIPs performance under various hardware configurations compared with the state-of-the-art systems and fine-tune WARCIP parameters, we also use a full-scale simulator to generate more performance results for the purpose of analyses and discussions. The simulator uses the same FTL design as the SSD prototype with page-level mapping for address translation and AVL-trees to maintain valid page number, and erasure count for each physical block. The write amplification factor and erasure count are accurate in our simulator since its Flash-Transfer-Layer design is the same as our SSD prototype. Two AVL-trees are used in the FTL: One maintains used blocks based on their valid page counts and the other maintains free blocks ordered based on their ages (erasure counts). Whenever an open block is full, the simulator chooses the youngest block which has the least

erasure count as the new open block. 8 open blocks are supported in the simulator. When the number of free blocks is lower than a threshold, which is 6 in our experiments, the simulator grabs the block that has the least valid pages as the erase candidate. To analyze WARCIP under heavy GC loads, we use 1% OP space in the simulator to minimize the impact of OP space.

The hot-cold data classification method [20] is implemented and used as baseline (BL) for our performance comparison purpose. BL distinguishes user write requests from GC write requests and stores them to different blocks accordingly. PCStream [11], FStream [10] and AutoStream [9] are the three state-of-the-art algorithms of I/O streams separation. We chose AutoStream for comparison because AutoStream is a block-level solution same as our work, while PCStream and FStream leverage system call or filesystem that cannot be integrated into SSD. AutoStream's Sequentiality Frequency Recency (SFR) algorithm is implemented both in the simulator and our SSD prototype for comparison purpose. We implemented SFR based on Algorithm 2 presented in [9] and optimizations of pre-defined parameters of SFR such as $decay_{period}$ to make sure SFR gets its best results in our experimental platform.

Workloads: Standard I/O benchmarks and I/O traces from real-world I/O workloads are used in our experiments. The first benchmark we used is TPC-C [21]. TPC-C is a well-known benchmark used to simulate the execution of a set of distributed and online transactions (OLTP). That TPC-C combines five types of transactions and nine types of tables that generate I/O requests with different complexities makes it a good benchmark for pattern-based evaluations.

Since the objective of WARCIP is to minimize rewrite interval variance of pages in a physical block. Traces with different I/O access patterns are essential. Therefore, we choose the traces from Microsoft Research Cambridge (MSRC) data

center [22]. These I/O traces were collected from 13 typical servers in the data center and covered a period of one week (168 hours). We believe that MSRC traces are the most suitable trace group available today to evaluate WARCIP for its diversity in I/O patterns and length in execution times. Before starting each measurement experiment using the I/O traces, we warm up the SSD by sequentially writing the entire SSD once so that our experiments were not started on an empty SSD. To minimize the influence of filesystem, all traces are sent to SSD using direct I/O.

1.5 Results and Discussions

In this section, we report and discuss our experimental results to demonstrate the effectiveness of WARCIP. Performance analysis and comparison will be carried out in terms of write amplification, erasure count, I/O performance, and additional resources required.

1.5.1 Write Amplification Reduction

To quantitatively evaluate WA reduction of WARCIP, we first run TPC-C benchmark and MSRC traces on our SSD prototype. TPC-C was set up and run for 6 hours on the SSD using 1750 warehouses (∼157GB in size) and 128 clients. TPC-C run on MySQL database with InnoDB as its storage engine. The buffer pool size of InnoDB is 128MB. Thus, a large portion of the database is located in SSD (storage-resident). To enable WARCIP to work on the SSD, the number of open blocks was configured as 4 and 8, respectively. The Write Amplification Factors (WAF) of TPC-C experiments are shown in Figure 4. The WAF of the BL SSD is 1.71 times larger than that of WARCIP with 8 open blocks. Notice that our SSD prototype is still under development. Its current version is pre-alpha with strict GC and user write flow control to allow the prototype to run reliably. The

flow control restricts the ratio of GC write requests to user write requests to be less than 2, which means the maximum WAF is limited to 3. We expect that the WAF reduction of WARCIP will be much larger once this restriction is removed when our SSD stabilizes. It is also observed in our experiments that the more open blocks configured, the better performance obtained. As shown in Figure 4, WARCIP with 8 open blocks shows lower WAF than with 4 open blocks. These results indicate that WARCIP can utilize open blocks well to achieve higher WAF reduction. Without loss of generality, we used 8 open blocks in all of the following evaluations.

In order to see how WARCIP reduces WA in real-world I/O workloads, we use MSRC traces to drive our experiments. Three write intensive MSRC traces *prn_1*, *proj_1* and *usr_1* are fed to our SSD prototype after warming up the SSD as mentioned in Section 1.4. The measured results are listed in Table 1. For all three cases, WARCIP reduces WA dramatically. The extra data migrations of traces *prn_1* and *proj_1* driven tests are less than 1MB, while they are over hundreds to thousands of megabytes on the baseline SSD. Even the maximum WA of WARCIP-SSD (*usr_1*) is less than one-tenth of that of the BL SSD. Fewer data migrations mean less back-end bandwidth consumption that can otherwise be used to serve user I/O requests. As a result, the latency of user I/O requests can be reduced, and I/O throughput is increased. This will be further discussed in later part of this section.

Besides comparing our WARCIP with the baseline SSD, we have also com-

Table 1: Write Amplification of three MSRC traces (The amount of data copied during GC operations in megabytes).

	prn_1	**proj_1**	**usr_1**
BL	**1459.250**	**863.668**	**7123.0**
WARCIP	**0.023**	**0.441**	**554.9**

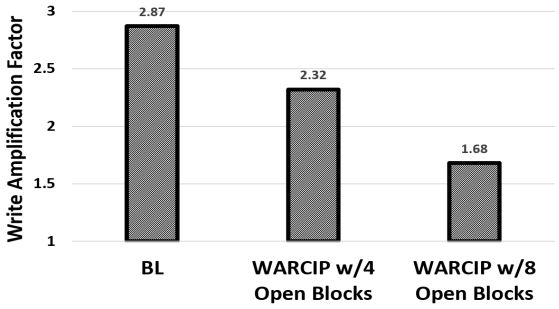


Figure 4: SSD WAF of WARCIP under different open block numbers in TPC-C benchmark (The lower, the better).

pared it with the most recent work on minimizing WA, specifically AutoStream [9]. Since the clustering algorithm of WARCIP is time-dependent, the 36 one week-long MSRC traces may take 36 weeks to complete for one configuration. Therefore, we ran all 36 MSRC traces on our simulator and measured WAs of WARCIP and AutoStream (AS), and compared them with the baseline SSD. We plotted the WA improvement of WARCIP and AutoStream relative to the baseline for all 36 I/O traces as shown in Figure 5. While both WARCIP and AutoSteam reduce WA of the I/O traces, WARCIP did a much better job than Autostream in all cases. For two-thirds of the 36 traces, WARCIP is able to improve WA by over 4 times as compared to the baseline SSD. The average WA improvement of WARCIP over the baseline SSD is 4.46 times while AutoStream is 2.79 times. Although the im-

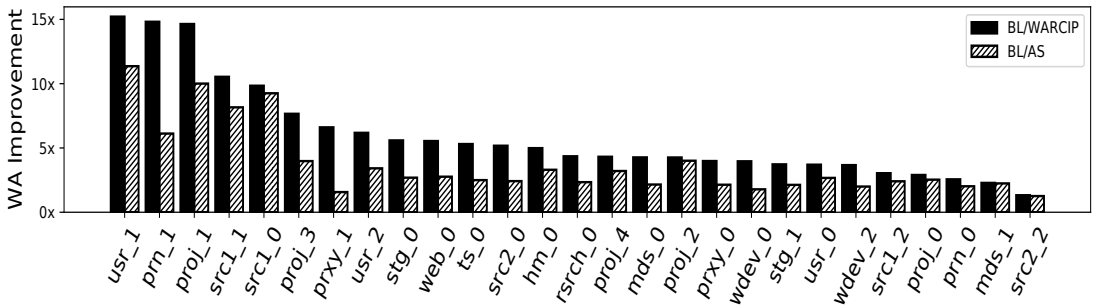


Figure 5: SSD Write Amplification improvement in MSRC traces (The higher, the better).

provements vary across different workloads, WARCIP achieves better results than AutoStream in all situations without exception. These consistent performance improvements can be mainly attributed to the fact that WARCIP is adaptive and self-optimized to different kinds of workloads.

1.5.2 Erasure Count Reduction

The lifetime of a flash memory SSD is mainly determined by its PE cycles, or erasure counts. Lower WA means fewer data pages being written to flash giving rise to fewer block erasures. In other words, WARCIP can increase the endurance of SSD by having less GC and erasure operations. For example, If WARCIP reduces erasure counts by half, the SSD's lifetime is likely doubled under the same I/O workloads. To illustrate this, we collect the erasure counts of WARCIP and the baseline SSD on our simulator driven by all 36 MSRC traces, and plot them in Figure 6 (read dominated traces that do not trigger erasure operations are not shown here). For better illustration, we divide the results into two groups with different scales. The trace group shown on the left-hand side of Figure 6 has higher write intensity than the group shown on the right-hand side of the figure. Since WARCIP is a learning algorithm that needs write requests to train its clustering model. More write requests give better model accuracy. As shown on the left-hand side of Figure 6, WARCIP reduces the erasure counts up to an order of magnitude

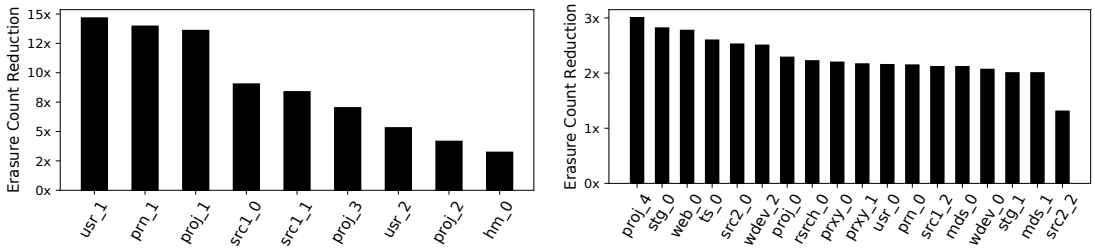


Figure 6: SSD erasure count reduction (BL/WARCIP) by WARCIP in MSRC traces (The higher, the better).

for some traces. The overall average reduction of erasure counts of all traces shown in the figure is 4.45 times as compared to the baseline SSD.

1.5.3 Performance

As mentioned previously, GC and WA have a negative impact on front-end I/O performance. Reducing WA can help improving I/O performance. We measured the average I/O throughput of our prototype SSD equipped with WARCIP (WARCIP-SSD) per 10 seconds while running TPC-C benchmark and compared it to AutoStream and the baseline SSD. TPC-C benchmark is set up to run 900 seconds with 60 seconds to warm up, and using 1750 warehouses and 128 clients. Please note that our SSD prototype is still under development, our main objective here is to show the relative performance under the same hardware conditions. The absolute throughput value will be higher after our SSD prototype is fine-tuned. Figure 7 plots the TPC-C experiment results. At the beginning, WARCIP, AutoStream (AS) and the baseline (BL) SSDs have similar throughput around 80MB/s. The throughput of AS is increased to 95MB/s after the 300th second, while BL remained around 80 MB/s. WARCIP accelerated when it learnt the I/O patterns around the 180th second, and its throughput increased gradually to 100MB/s. One may notice that there is a peak throughput of the baseline SSD at around the 150th second and a slowdown at around the 630th second of WARCIP-SSD and AS-SSD. To understand these abnormal results, we look into the log of the SSD activities during our experiments. During the period around the 150th second, no GC is triggered since there are enough free blocks in the system. All back-end bandwidth is used by the front-end I/Os, giving rise to a high user I/O throughput. This also explains the small peaks throughout the tests. The performance drops of WARCIP and AutoStream are caused by database's internal processing during TPC-C test. The internal activities, such as log flush, changed

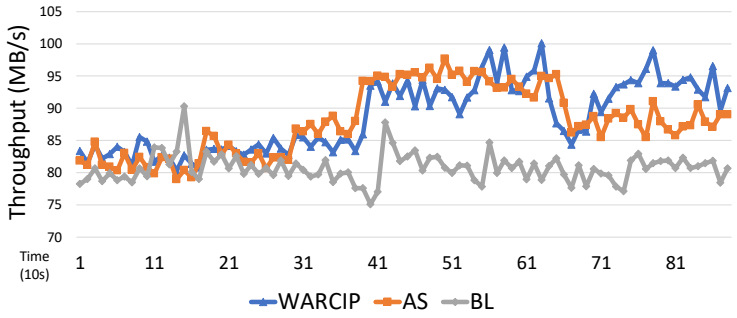


Figure 7: SSD throughput of TPC-C benchmark (The higher, the better).

I/O patterns by increasing queries response time and causing poor grouping results. However, with the D-SAM mechanism, WARCIP is able to adapt to the new pattern in a short period. As shown in Figure 7 from the 700th second to the 800th second, the throughput of WARCIP goes up again after self-optimization while the throughput of AutoStream SSD drops below 90MB/s because it can not adapt to the new I/O patterns automatically.

Besides throughput, we have also measured the response time of three write intensive MSRC traces on our SSD prototype. Table 2 shows the average response times of three traces running in all SSDs. In all cases, WARCIP is much more efficient than AS in terms of reducing read and write response times. We noticed that AS did not show much better performance than the BL for *prn*_1 trace because of its lack of adaptivity to changing I/O patterns. The best improvement occurs on the *usr*_1 trace. WARCIP reduced read and write response times by 29.1% and 65.0% while AS showed 11.2% and 51.6% improvement, respectively. The *usr*_1 trace contains the most I/O requests (over 45 million) among all the three traces. This indicates that WARCIP can achieve better performance after learning from more I/O requests. Table 2 also reveals that write requests are served faster and gain more improvement than read requests. This is due to the internal mechanism of our SSD prototype. In our SSD prototype, write requests are written to a write

Table 2: I/O Response Time (μs) of three MSRC traces (Less response time indicates better result).

	prn_1			proj_1			usr_1		
	BL	AS	WAR-CIP	BL	AS	WAR-CIP	BL	AS	WAR-CIP
Read	1023	1019	825	1434	1206	1067	2548	2261	1807
Write	528	540	484	389	328	208	985	477	345

buffer and returned, then the SSD controller programs these requests' data to different flash dies in parallel to exploit SSD's internal parallelism. On the other hand, read requests are directly served by the die that contains requested data. Therefore, read requests are more likely to be blocked by programming or erase operations, making it difficult to take advantages of the back-end bandwidth that is released by WARCIP. What is more, when blockage happens, SSD firmware tries to serve all write commands alternatively giving rise to higher priority for write requests to be served.

To further illustrate WARCIP's advantages on reading blockage reduction over the baseline SSD, we ran all MSRC traces in the simulator and measured the number of blocked read requests under WARCIP-SSD and the baseline SSD, respectively. As shown in Figure 8, WARCIP reduces the number of blocked read requests significantly (Y-axis is in log scale). For some traces such as *prn_0*, WARCIP does not suffer from any read blockage but the baseline SSD does. The results also prove that WARCIP can reduce SSD's internal operations and improve the service quality of read requests, which is important for enterprise applications.

1.5.4 Overhead Analysis

As mentioned in Section 1.2.2, WARCIP needs to maintain a time stamp for every 8 consecutive logical pages and an average RWI for each open block. In our current implementation, each time stamp takes 4 bytes to record a maximal inter-

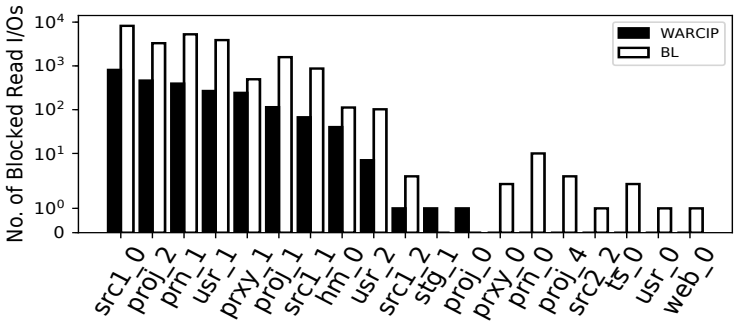


Figure 8: Improvement on blockage of read I/Os (The less, the better).

val time of 25 days. The total DRAM required for this purpose is 1GB DRAM for our 8TB SSD prototype. In comparison, SFR needs 16 bytes per chunk. To achieve the same level of precision as WARCIP (8 consecutive logical pages per unit), AutoStream takes 4GB DRAM space, 4 times larger than that of WARCIP. Regarding CPU usage, the RWI calculator and Clustering decision maker of WARCIP contains a few simple mathematical calculations and comparisons. The average CPU time used for each clustering decision is less than $400ns$ which is unnoticeable in practice.

1.6 Related Work

Because of its importance, extensive research has been reported in the literature to reduce the negative impact of GC in flash memory SSDs [23, 24, 25] and to cluster streaming data for data mining [26, 27]. For a comprehensive review, readers are referred to [15, 28, 29]. In this section, we discuss prior researches that are closely related to our work.

Write amplification reduction: SDF [30] and LOCS [31] decrease write amplification by using large write unit (2MB to 8MB). They are designed for LSM-tree based workloads and do not work well on small writes dominated applications such as online transaction processing (OLTP). WA can also be reduced by invaliding pages ahead of GC [2] or only copying pages which do not have modified

copies in main memory during a GC operation [3]. However, either way needs the upper layer cache to pass cache information to SSD controller. This requires the modifications in the operating system, which limits the usage. Other researches use greedy reclaiming policy [4, 5, 6] to reduce WA. They choose the block that has the least valid pages for GC. Although greedy reclaiming policy achieves minimum data migration during each GC, it cannot reduce lifetime variance of pages in that block. To eliminate the root cause of WA, separating data according to their RWIs is the way to go, which has been our focus in this work.

Another approach is to provide more free space [32, 33, 34, 35]. Large over-provisioned space increases GC trigger intervals and allows more pages to be invalidated in a block before being erased. Over-provisioning improves overall garbage collection performance since fewer data migrations are required. However, larger over-provisioning means less storage space available to users or higher cost of SSD. This is clearly not a cost-effective approach.

Write Interval Detection and Separation: SFS [7] separates "hot" and "cold" data to different segments in the buffer cache (DRAM) and writes them to SSD in batches. Nevertheless, SFS defers "small" writes in volatile memory until a segment is filled completely. It does not benefit applications such as the database that requires data to be persistent after a checkpoint or the completion of a flush operation. ETI [36] divides the address space into N hot/cold clusters based on extents. The computation complexity for the ordered search structure may not be suitable for high-end enterprise SSDs that require extremely high IOPS. Multi-Stream SSD [8] exposes open blocks to upper layer applications and let applications choose which block to use. In this way, applications put data with similar life-cycles in the same physical block to reduce write amplification. PCStream [11] automates the stream selection based on program counters in the

Linux kernel. FStream [10] identifies journal and metadata of file systems and maps these data to different streams. AutoStream [9] is transparent to applications for it detects I/O frequency in Linux device driver and divides writes into streams based on predefined parameters. While PCStream, FStream and AutoStream require modifications in the operating system, WARCIP is designed for working inside SSD and can plug and play without any modification of applications and the operating system. It should be also noted that the major distinctive features of WARCIP are its adaptivity to dynamic changing I/O workloads and self-optimization through a reward mechanism. As a result, no parameter tuning is needed for WARCIP to fit to different workloads.

Clustering: The k-means algorithm [37] is one of the most famous clustering algorithms. It calculates k cluster centers iteratively and assigns objects to their nearest centers by measuring the Euclidean Distance between the objects and cluster centers. k-means++ [38] improves the initialization procedure of k-means by selecting cluster centers that are far from each other. k-means++ runs faster than k-means and provides better results. Inspired by k-means++, Ailon et al. provide k-means# [17]. k-means# chooses a subset of the stream in a single pass and achieves a constant approximation to the k-means objective. StreamKM++ [18] is a two-step stream clustering algorithm based on K-means++. StreamKM++ uses coreset tree and bucket to reduce-and-merge objects. Although the above clustering algorithms can cluster I/O stream, they are not suitable for reducing WA in SSD. To reduce WA, the cluster algorithm needs to have fixed cluster size which is equal to the block size and the number of clusters is unknown in advance. Our proposed WARCIP is designed with these constraints in mind. WARCIP has fixed cluster size with unlimited cluster number. WARCIP can adjust its clustering strategy to fit the changing I/O patterns and change the number of clusters

for the purpose of better utilizing open block resources.

1.7 Conclusion

In this paper, we proposed WARCIP, a new approach to minimizing the negative impact of garbage collection on SSD's I/O performance and endurance. WARCIP groups pages into a block according to their rewrite intervals to reduce write amplification caused by page migrations during garbage collection operations. Through learning the I/O patterns of write streams, WARCIP can reduce rewrite interval variance of pages in a physical block giving rise to less write amplification. We have implemented a fully functional WARCIP on an NVMe SSD card. Both simulation and real measurement experiments have been carried out on WARCIP using real-world I/O traces as well as standard benchmarks. Experiment results show that our WARCIP can reduce write amplification dramatically and decreases the number of block erasures by up to an order of magnitude.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments that helped greatly in improving the quality of the paper. The authors are thankful to Ying Yang, Shaoquan Liu and Yuanpeng Ma for providing guidance to this work. This research is supported in part by the NSF grants CCF-1439011 and CCF-1421823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. It is also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd.

List of References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[2] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvm," *ACM Trans. Storage*, vol. 13, no. 2, pp. 12:1–12:13, May 2017.

[3] G. Xu, M. Wang, and Y. Liu, "Swap-aware garbage collection algorithm for nand flash-based consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 1, pp. 60–65, February 2014.

[4] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system." in *USENIX*, 1995, pp. 155–164.

[5] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[6] L. Han, Y. Ryu, and K. Yim, "Cata: a garbage collection scheme for flash memory file systems," in *International Conference on Ubiquitous Intelligence and Computing*. Springer, 2006, pp. 103–112.

[7] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: random write considered harmful in solid state drives." in *FAST*, vol. 12, 2012, pp. 1–16.

[8] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 13–13.

[9] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Autostream: Automatic stream management for multi-streamed ssds," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 3:1–3:11.

[10] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J.-Y. Hwang, S. Cho, D. D. Lee, and J. Jeong, "Fstream: managing flash streams in the file system," in *16th USENIX Conference on File and Storage Technologies*, 2018, p. 257.

[11] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, "Fully automatic stream management for multi-streamed ssds using program contexts," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 2019, pp. 295–308. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kim-taejin

[12] Y. Bu, H. Lee, and J. Madhavan, "Comparing ssd-placement strategies to scale a database-in-the-cloud," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 41.

[13] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, *et al.*, "Schema-agnostic indexing with azure documentdb," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1668–1679, 2015.

[14] N. Zhang, J. Tatemura, J. Patel, and H. Hacigumus, "Re-evaluating designs for multi-tenant oltp workloads on ssd-basedi/o subsystems," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 1383–1394.

[15] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 15–28.

[16] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014. New York, NY, USA: ACM, 2014, pp. 7:1–7:11.

[17] N. Ailon, R. Jaiswal, and C. Monteleoni, "Streaming k-means approximation," in *Advances in neural information processing systems*, 2009, pp. 10–18.

[18] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++: A clustering algorithm for data streams," *Journal of Experimental Algorithmics (JEA)*, vol. 17, pp. 2–4, 2012.

[19] N. E. Workgroup, "Nvm express specification 1.3 a. http://www. nvmexpress. org/specifications," 2017.

[20] B. Van Houdt, "On the necessity of hot and cold data identification to reduce the write amplification in flash-based ssds," *Performance Evaluation*, vol. 82, pp. 1–14, 2014.

[21] TPC-C. "tpcc-mysql benchmark." 2017. [Online]. Available: https://github.com/Percona-Lab/tpcc-mysql

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 10, 2008.

[23] M. Jung and J. Yoo, "Scheduling garbage collection opportunistically to reduce worst-case i/o performance in solid state disks," *Proceedings of IWSSPS*, 2009.

[24] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension." in *FAST*, vol. 12, 2012, pp. 10–10.

[25] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 12–21.

[26] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 81–92.

[27] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 133–142.

[28] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. De Carvalho, and J. Gama, "Data stream clustering: A survey," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 13, 2013.

[29] M. Ghesmoune, M. Lebbah, and H. Azzag, "State-of-the-art on clustering data streams," *Big Data Analytics*, vol. 1, no. 1, p. 13, 2016.

[30] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 471–484.

[31] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 16:1–16:14.

[32] R. Lucchesi, "Ssd flash drives enter the enterprise," *Silverton Consulting. accessed on*, vol. 8, p. 2008, 2011.

[33] G. Drossel, "Methodologies for calculating ssd useable life," in *Proc. Storage Developer Conf*, 2009.

[34] Seagate. "Ssd over-provisioning and its benefits." 2019. [Online]. Available: https://www.seagate.com/tech-insights/ssd-over-provisioning-benefits-master-ti/

[35] Kingston. "Understanding over-provisioning (op)." 2019. [Online]. Available: https://www.kingston.com/us/ssd/overprovisioning

[36] M. Shafaei, P. Desnoyers, and J. Fitzpatrick, "Write amplification reduction in flash-based ssds through extent-based temperature identification," in *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[37] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[38] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

MANUSCRIPT 2

Thermo-GC: Reducing Write Amplification by Tagging Migrated Pages during Garbage Collection

Jing Yang[1], Shuyi Pei[2]

is accepted by the 14th IEEE International Conference on Networking,

Architecture and Storage.

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: jyang@ele.uri.edu

[2]Graduate Student, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: spei@ele.uri.edu

Abstract

Flash memory based solid-state drive (SSD) has been deployed in various systems because of its significant advantages over hard disk drive in terms of throughput and IOPS. One inherent operation that is necessary in SSD is garbage collection (GC), a procedure that selects an erasure candidate block and moves valid data on the selected candidate to another block. The performance of SSD is greatly influenced by GC. While existing studies have made advances in minimizing GC cost, few took advantages of the procedure of GC itself. As GC goes on, valid pages in an erasure candidate block tend to have similar lifetimes that can be exploited to minimize page's movements. In this paper, we introduce Thermo-GC. The idea is to identify data's hotness during GC operations and group data that have similar lifetimes to the same block. By clustering valid pages based on their hotness, Thermo-GC can minimize valid page movements and reduce GC cost. Experiment results show that Thermo-GC reduces data movements during GC by 78% and write amplification factor by 29.7% on average, implying extended lifetimes of SSDs.

2.1 Introduction

NAND flash based solid-state drives (SSDs) have swept the storage market in the past few years. They have continually replaced conventional magnetic hard disk drives (HDDs) from data centers to consumer devices due to SSD's increased performance and reduced cost. Unlike conventional HDD which partitions data into sectors (e.g., 512B), data in an SSD is organized in blocks and each block contains a fixed number of pages (e.g., 256 pages per block and 4KB per page). Because of NAND flash's physical characteristics, a page needs to be erased before it is written again. Typically, the erasure unit of an SSD is a block. Before erasing, valid data in that block need to be moved out. The process of selecting an erasure candidate block, migrating valid data pages, and erasing the block is known as garbage collection (GC).

GC is a time-consuming process, the data migration caused by GC consumes a significant amount of the back-end bandwidth of an SSD and increases the response time of user I/O requests. The cost of GC becomes higher when an SSD is filled with more data because GC is triggered more frequently. The additional write operations caused by GC is known as write amplification (WA).

Numerous studies have been carried out to mitigate WA and reduce GC count [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. One approach is to invalid page ahead when the page has a copy or update in the upper layer cache (e.g., [1, 2]). Since these kinds of pages will be updated in SSD in a short time, the efforts of GC may soon be vain. Another approach is wisely choosing the GC candidate block that has less valid pages while considering the age of data to minimize data migrations at the same time (e.g., [3, 4, 5, 6]). WA can be further reduced by separating hot and cold data [7, 8, 9, 10]. This helps to reduce the number of valid pages in an GC candidate block because pages in a hot block are invalidated in a shorter time

frame.

While existing research has made advances in terms of minimizing WA and reducing GC count, few take advantage of GC procedure itself to reduce WA. There are several advantages to identifying data lifetimes during GC. First of all, identifying data's lifetime during GC does not introduce delay to the user I/O processing path that is time sensitive. The new NVME-SSD processes hundreds of thousands of user I/Os per second. Since the computing power of SSD's embedded processor is limited, any additional processes on the user I/O processing path can slow down the throughput of an NVME-SDD by tens of thousands I/Os per second. Secondly, GC candidate block's lifetime can be used to estimate valid pages' lifetime since valid pages have longer lifetimes than the block. This can reduce the required memory resource for pages' lifetime identification. For example, a 1TB SSD contains 256 million 4KB pages requiring 1GB DRAM to maintain a 32-bits timestamp for every page. The memory cost can be reduced to 4MB (256 pages per block) if we use the lifetime of a block to estimate the lifetimes of valid pages in that block. Finally and most importantly, most existing hot/cold data identification methods employ fixed length data structure such as multi queues or lists [7, 8, 11, 12, 13]. Thus, only a fixed number of hot pages can be identified. However, the number of hot pages varies for different applications. An oversized or undersized hot page pool will increase the lifetime variance of pages in a block and result in more WA during GC.

To tackle these challenges, we introduce Thermo-GC. Thermo-GC identifies pages' hotness by leveraging valid pages' recency and frequency information during GC. Thermo-GC provides an online learning process that can adapt to the changing workload patterns. To further reduce WA, Thermo-GC also separates user requests based on the grouping information obtained from GC. By taking advantages of the

SSD's page invalid process, Thermo-GC can group user write requests as a free ride of the page invalid process and reduce the memory resource requirement for classification. We will discuss this in details in Section 2.3.3. To quantitatively evaluate the performance of Thermo-GC, we implemented our Thermo-GC in the new open source SSD simulator named MQSim [14] which can accurately model the performance and internal activity of modern SSDs.

In this paper, we made the following key contributions:

- We provide in-depth experimental analyses of the GC behavior of an SSD in the steady-state. We found that valid pages are often moved more than once, which is a major source of WA when an SSD has high occupancy.

- We propose Thermo-GC, a page hotness identifier. Thermo-GC leverages the valid page information in a GC candidate block to estimate pages hotness and uses this information to reduce pages' lifetime variance in a block.

- Thermo-GC is resource efficient and cost effective. It achieves page-level grouping accuracy by only using a per block group number, which reduces the memory resource requirement for page classification. Moreover, Thermo-GC clusters user write requests during SSD's page invalid process as a free ride.

- Thermo-GC has been implemented in a standard SSD simulator. Comprehensive experimental evaluations have been carried out to show the advantages of Thermo-GC over the state-of-the-art techniques that identify data's lifetimes inside SSD.

The remainder of this paper is structured as follows. Section 2.2 provides background and our extended motivation. Thermo-GC's design and implementation are given in Sections 2.3 and 2.4, respectively. Section 2.5 describes our

simulator platform and the workloads that are used to evaluate Thermo-GC. Experiment results are presented and discussed in Section 2.6. Related works are discussed in Section 2.7 and we conclude our work in Section 2.8.

2.2 Background and Motivation

This section first briefly review the necessary background, followed by discussions on experiments that motivate our design.

2.2.1 Inside SSD

Modern SSDs use NAND flashes to store data. NAND flash is a kind of nonvolatile memory that is read or written in the unit of a page, but erases at the granularity of a block. There are hundreds of pages in each block. Since NAND flash does not support in-place update, modern SSD uses relocate-on-write strategy. As shown in Figure 9a, whenever the data in a page is updated, SSD simply marks the page as invalid and writes the data to an open block that has been erased. An open block is the appending point of a flash that receives program (write) requests. Therefore, a flash page can be in one of free, valid, or invalid status, indicating the page is erased, contains valid, or obsolete data. A block can be in free, valid, opened, or GC candidate status, indicating the block is erased, fully programmed, partially programmed, or under GC.

GC is the procedure of SSD that collects valid pages and discards obsolete data. GC is triggered when the number of free blocks falls below a threshold. First, GC chooses a valid block as GC candidate block (e.g. the block that contains the least valid pages). Second, GC examines each page in the GC candidate block and reads the data of valid pages out of the block. Finally, these data are written to free pages in the open block as shown in Figure 9b. The data migration caused by GC consumes a lot of internal bandwidth of the SSD and increases the response

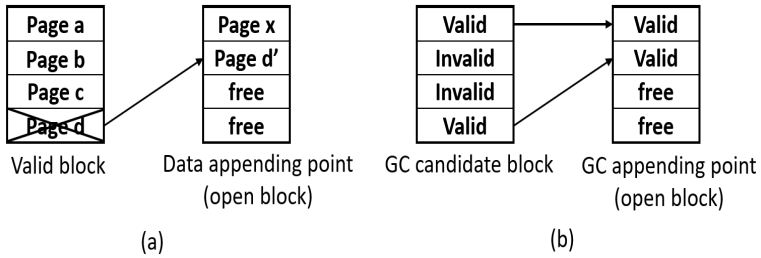


Figure 9: Out-of-Place update and garbage collection inside SSD.
(a) When the data in page d is updated, new data are written to a free page in the open block as page d', and the old page d is invalidated. (b) All valid pages in the GC candidate block are moved to open block before the GC candidate is erased.

time of user I/Os, especially when SSD is full of data. We will discuss this in the next section.

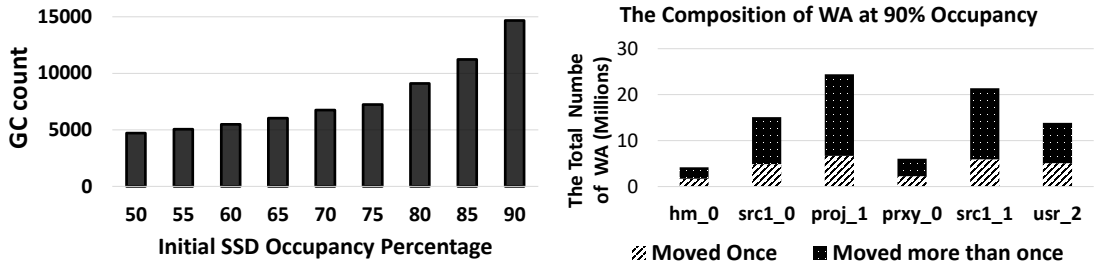
2.2.2 An Experimental Analysis of GC Activities

In this section, we explain the impacts of GC and how they motivate our work by two experiments. We use MQSim [14] to carry out our experiments. In our experiments, MQSim is configured with 2 channels, 1 chip per channel, 2 dies per chip and 2 planes per die. Both synthetic workloads and real world traces are run through MQSim for evaluations.

In the first experiment, we investigate how different usages of SSD impact the GC activities of an SSD. One million write requests are sent to the simulator randomly with different initial occupancies. 70% initial occupancy means 70% of the storage space is filled before the experiment. Nine kinds of data occupancy are tested in the experiment. The results in Figure 10a indicate that GC becomes more active with the increase of data occupancy. For example, the number of instances of GC is 4725 when the initial data occupancy is 50%. When the initial data occupancy is 90%, the total GC count is increased over 3 times to 14657. The reason is that the more data an SSD is filled with, the less invalid pages can be found in a GC candidate block. Each GC releases less space that results in more

GC activities.

To further analyze the GC behavior of an SSD at high occupancy, six Microsoft data center traces [15] are fed to MQSim. The initial occupancy of the simulated SSD is set to 90%. We collected the WA induced by valid pages in terms of "moved once" and "moved more than once" separately through the whole trace running process. "Moved once" means a valid page is only moved once before it is updated (invalid). "Moved more than once" indicates a valid page is moved twice or more times before it is updated. The results are shown in Figure 10b. From the figure we can see that valid pages that are moved twice or more times by GC contribute to the major part of the total WA when an SSD has high occupancy. The proportions of twice or more valid page movements are more than 50% in all six traces. We also noticed that the proportion of twice or more valid pages movements are higher when a trace contains more write requests. This can also be observed in other traces from the Microsoft data center. The twice or more valid pages' movements occupy a large amount of back-end bandwidth and further reduce the endurance of an SSD. It is crucial to mitigate this kind of WA to improve front-end response time (e.g. user I/O response time) and SSD lifetime.



(a) (b)

Figure 10: (a) GC activities of an SSD on different initial occupancies. (b) The Composition of WA at 90% initial occupancy of an SSD.

2.3 Thermo-GC design

Based on the analyses above, we will explain how to minimize WA by leveraging GC activities of an SSD in this section.

2.3.1 Lifetime estimation

Since SSDs are shared by many applications in the cloud or virtual machine environments, write requests that have different update cycles are tangled and stored in the same flash block. Therefore a fine-grained page level measurement is necessary to reduce the lifetime variance of a block. Frequency and recency are the two major features that are used in lifetime identification. They have been proven to work well in storage systems such as caches. However, both frequency and recency introduce significant overhead on user I/O requests due to the limited computation power and DRAM resource inside SSD. To track frequency information, a counter is needed to maintain frequency information for each page and a decay method is required to prevent counter from overflow. At least a timestamp or a position indicator of a list is required per page for recency tracking. Both of them cost a large amount of DRAM resource since the storage space of an SSD becomes very large nowadays. For example, a 4 bytes timestamp per 4KB page costs 1GB RAM for a 1TB SSD. Even in the modern high-end SSD, there is only 2 to 4 GB DRAM and most of them are used for the address translation. So we need to exploit the DRAM efficiently for data lifetime identification. Moreover, computational overhead is another important issue since it has to be triggered whenever a write request is issued, it is expensive to use complicated lifetime identification algorithms inside SSD.

Although the unavoidable GC activities of SSDs take up a lot of back-end bandwidth and reduce the endurance of an SSD, it provides us with the lifetime information of its pages. The valid pages in the erasure candidate block indicate

their lifetimes (update intervals) are longer than invalidated pages in that block and the block itself. As shown in Figure 11, a block is opened (receives write requests) at time point α, fully programmed at time point β and selected as GC candidate at time point γ. Pages a and d are still valid at time point γ. These valid pages have longer lifetimes than the block itself whose lifetime is γ minus β . This motivates us to use the block lifetime to estimate the valid pages lifetimes in a block. Thermo-GC calculates a block's lifetime by subtracting the recorded time stamp of the block when it is fully programmed from the time it is selected as the GC candidate block. Whenever a block is selected as the GC candidate, all valid pages in the block are marked as they have longer lifetime than the block by using the block's lifetime.

2.3.2 Valid Page Clustering

As discussed in Section 2.1, most of the existing hot/cold data identification techniques can only identify a small fixed number of hot pages. However, the size of hot pages changes for different applications, and the boundary between hot and cold data is not that obvious for real workloads. To tackle these challenges, Thermo-GC uses a two-level cluster strategy to separate valid pages to different

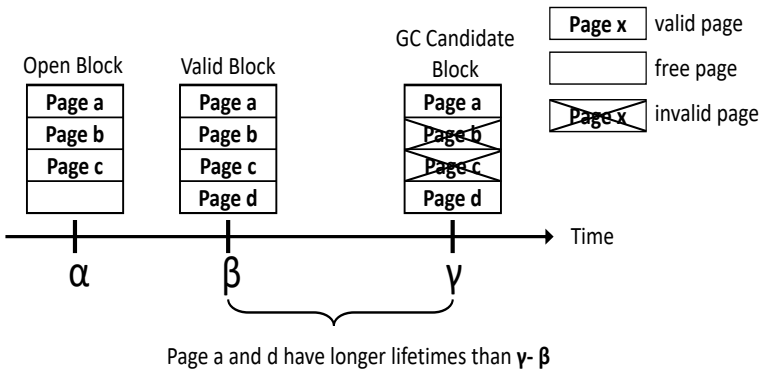


Figure 11: The lifetime measurement of a page. The lifetimes of page a and d can be estimated by γ minus β.

blocks by taking recency and frequency into account as shown in Figure 12. The idea comes from our observations in Section 2.2.2 that the valid pages in a GC candidate block may be GCed more than twice when the SSD has high occupancy. We refer to pages that have been moved more than once as *secondary WA*. Since *secondary WA* consists of a large portion of the whole GC activities, the first priority of Thermo-GC is to minimize the *secondary WA*. *Secondary WA* is caused by mixing valid pages from different GC candidate blocks into the same open block. *Secondary WA* can be reduced by separating valid pages to different blocks according to their hotness. To achieve this, we introduce multiple GC appending points. By increasing the number of GC appending points, valid pages can be clustered to different blocks according to their estimated lifetimes.

The clustering results need to be adjusted at run-time since workloads vary from time to time. We maintain a sliding window to monitor recent GC activities. The sliding window maintains the estimated lifetime of each recent moved valid page. We further divide recorded lifetimes of pages in the window to B groups based on the pages' lifetimes. Whenever a valid page is read from the GC candidate block, Thermo-GC assigns the page to one of the groups according to its lifetime's position of the ordered lifetimes in the sliding window. Then, the page is written to different GC appending points based on the assigned group number.

If pages have been moved more than once by GC, they usually have long lifetimes (update intervals), and their lifetimes are scattered over a large time range. Clustering them by lifetimes may reduce the overall classification accuracy. One way to solve this is by adding more GC appending points. However, the lifetime variance of pages in a block may still be high since their lifetimes are scattered and a large number of appending points increases management overhead of the SSD which is not that practical.

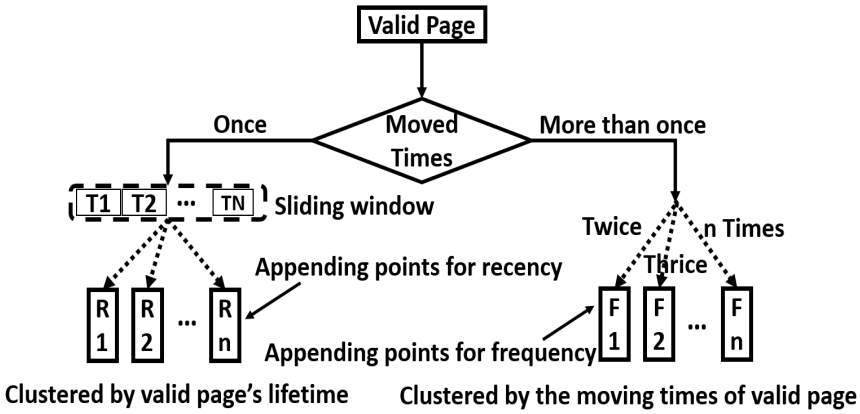


Figure 12: The valid pages clustering algorithm of Thermo-GC.

Since we can easily get the number of a page's migrations, it guides us to use frequency for cold pages' classification. Thermo-GC uses WA frequency instead of recency for lifetime identification of pages that are GCed more than once. When a valid page is read from a GC candidate block, Thermo-GC checks the number of GCs on this page. If it is the first time GC, Thermo-GC groups this page by recency as discussed above. Otherwise, Thermo-GC clusters the pages by frequency which is measured by the number of times they have been moved.

2.3.3 Maintaining Groups

In section 2.3.2, we discuss how to group valid pages during GC. However, user I/Os can not benefit from the GC grouping because the grouping information only exist after a page is GCed. If user I/O requests can leverage the group information obtained from GC, WA can be further reduced. A simple way to achieve this is to make the group information persistent by storing a valid page's group information in the global mapping table. When a user write request arrives, it can be grouped by querying their previous group information stored in the global mapping table. Since pages are clustered into a limited number of groups, only a few bits per page are needed. For example, if 8 open blocks are used to classify valid pages during GC, only 3 bits are needed to identify the group for each page.

The memory requirement for maintaining the group information can be further reduced by taking advantages of the page's invalid process. From Section 2.2 we know that the page that contains obsolete data needs to be invalidated ahead before new data of the same logical address is written into the SSD. During invalidating process, the metadata of the block that contains the page is also updated (e.g. the number of valid pages). This inspires us to store pages' previous group number in the block's metadata. Moreover, we can get the page's group information as a free ride of the page's invalid process. This does not increase the I/O response time of user requests on the critical path of SSD.

To achieve this, Thermo-GC adds a variable to store the group number in SSD's block management structure. The memory resource requirement is significantly reduced since the group number is a per block variable. For example, the memory resource requirement can be reduced to $1/256$ when a block contains 256 pages. The group number is initialized with a default group number and updated by Thermo-GC during GC. All block's in the SSD will get a group number eventually as GC going on.

2.4 Thermo-GC implementation

The implementation details of Thermo-GC are discussed in this section by taking into account the practical factors of real systems.

2.4.1 Clustering valid pages during GC

There are 8 GC appending points used by Thermo-GC to cluster valid pages in the GC candidate block. A time stamp is added to the flash block management structure which is 8 bytes in size. It is set to the system time when the block is fully programmed. Thus, the lifetime of a GC candidate block can be calculated by subtracting the recorded time stamp of the block from the current system time.

After a block is selected as the GC candidate block, its lifetime is added to the sliding window. The sliding window is maintained as a FIFO list that the oldest block's lifetime is moved out when the sliding window is full.

The first 4 of the 8 GC appending points are used for clustering valid pages that are moved for the first time. Particularly, when a valid page is moved for the first time, Thermo-GC clusters the valid page to one of the 4 GC appending point according to the order it is in the sliding window, where all elements are sorted by their lifetimes. For example, if the valid page's lifetime is greater than 30% of all valid pages' lifetimes in the sliding window, Thermo-GC groups the valid page to group 1 (zero-based index). The other 4 GC appending points are used for clustering valid pages that are moved more than once. Particularly, valid pages that are moved twice are sent to the 4th GC appending. Valid pages that are moved three and four times are sent to the 5th and 6th appending points, respectively. Pages moved more than four times are sent to the 7th appending point.

2.4.2 User I/O separation

Thermo-GC uses 3 bits per block to maintain the grouping results. When a block is filled by GC, the corresponding group number is stored in the block's metadata. Whenever a user write request arrives, SSD checks if the LPA of the request already exists in SSD's FTL. If not, Thermo-GC assigns the default group number, which is 3 in our implementation, to the write request. Otherwise, Thermo-GC gets the old page's group number during updating the metadata of the block and assigns it to the new request. Then, the write request is stored in one of the eight data appending points according to the assigned group number of the request. Please note that, unlike other hot/cold data separating methods that calculate and maintain the hotness information of user requests during processing them,

Thermo-GC separates user request as a free ride of a page's invalid process which makes the overhead of Thermo-GC is negligible.

2.5 Evaluation Methodology

We have implemented Thermo-GC on the open-source SSD simulator named MQSim [14]. MQSim models modern NVMe-based SSDs and can simulate the steady-state behavior of SSDs, which is important for GC studies. Table 3 summarizes the configurations that we used in our evaluations. The simulated SSD is configured with 4 channels, 2 chip per channel, 2 dies per chip and 2 planes per die. Each plane contains 2048 flash blocks, each of which holds 256 flash pages. Each page is 8192 bytes in size. The total size of the simulated SSD is 128GB. We use 7% space as over provisioning, which is necessary for a working SSD. To make the SSD quickly reaches its steady-state, we enable *preconditioning* with 90% initial occupancy using the algorithm provided by the simulator.

To evaluate Thermo-GC's performance under different scenarios, we choose the trace group from Microsoft Research Cambridge (MSRC) data center [15]. These I/O traces were collected from 13 typical servers in the data center and covered a period of one week (168 hours). MSRC traces are the best trace group

Table 3: Configuration of the simulated SSD

SSD Organization	Host interface: PCIe 3.0 User capacity: 128GB 4 channels, 2 chips per channel
Flash Communication Interface	ONFI 3.1 Width: 8 bit, Rate: 333 MT/s
NAND Flash Microarchitecture	8 KB page, 448 B metadata per page 256 pages per block, 2048 blocks per plane 2 planes per die
NAND Flash Access Latency	Read latency: 75 μs Program latency: 750 μs Erase latency: 3.8 ms

available for us since it contains various I/O patterns and covers a long period. All 36 MSRC traces are fed into the simulator for five times to simulate long-term activities of the SSD.

The hot-cold data classification method named DWF [16] is used in our evaluation for comparison purpose. DWF separates GC write requests from user requests and stores them to different flash blocks. The Sequentiality Frequency Recency (SFR) algorithm of AutoStream [17] is also implemented in the simulator for comparison purpose. AutoStream has multiple data appending points and separates user write requests to different appending points based on their update frequency with recency constraint. We implemented SFR based on Algorithm 2 presented in their paper [17].

2.6 Results and Discussions

In this section, we report and discuss our experimental results to demonstrate the effectiveness of Thermo-GC. Performance analysis and comparison will be carried out in terms of write amplification, erasure count, performance, and additional resources required.

2.6.1 Write Amplification Reduction

In order to see how Thermo-GC reduces WA in real-world I/O workloads, all 36 MSRC traces are used to drive our experiments. The SSD is warmed up by preconditioning before these traces are fed to the simulator. We collected the amount of WA (i.e. total valid pages moved by GC) and calculated their WA reductions as shown in Figure 13. The results of traces wdev_1 and wdev_3 are not shown in this figure since we did not observe WA in these two experiments. Both wdev_1 and wdev_3 are from a web development server, and their total write requests sizes are 5MB and 2MB, respectively. AutoStream and Thermo-GC do not work well

in read dominated workloads since they are write-request-driven algorithms. From Figure 13 we can see that Thermo-GC reduces WA in all traces without exception when compared to DWF and AutoSteam. Specifically, Thermo-GC reduces WA by 78% on average when compared to DWF and 75% on average when compared to AutoStream. AutoStream is not designed for reducing *secondary WA*. When *secondary WA* becomes the major source of WA, the overall improvement of AutoStream is limited. To illustrate this, we collected the number of valid pages that are moved more than once during GC, and calculated Thermo-GC's reductions over DWF and AutoStream as shown in Figure 14. Thermo-GC reduced over 90% *secondary WA* on 27 traces and also got high WA reduction on the other traces. We also noticed that Figure 14 and Figure 13 have very similar trends, this confirms *secondary WA* is a major source of WA when SSD becomes full which is 90% in our case and Thermo-GC can significantly reduce *secondary WA* based on valid pages' recency and frequency information during GC.

2.6.2 Erasure Count Reduction

As NAND flash density increases, modern 3D NAND flash memories only have 3000 and 1500 program/erase (P/E) cycles for MLC and TLC, respectively [18]. The extra P/E cycles caused by data migration can significantly decrease the endurance of SSD. By leveraging valid pages' recency and frequency information during GC, Thermo-GC successfully reduces WA and results in fewer block era-

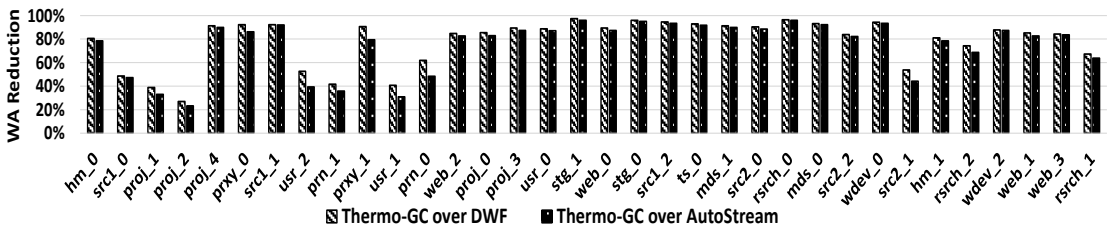


Figure 13: The WA reduction of Thermo-GC over AutoStream and DWF (The higher, the better).

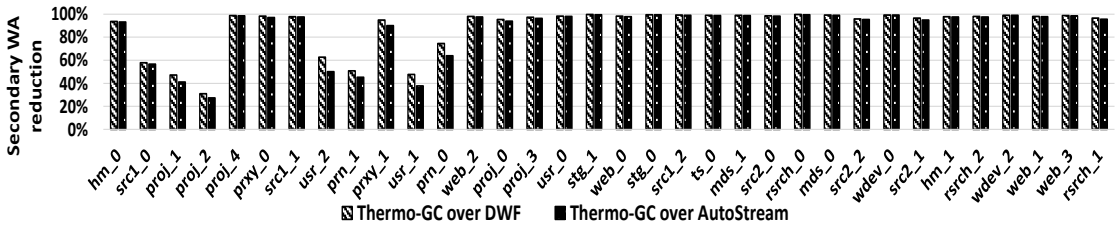


Figure 14: The reduction of the number of valid pages that are moved more than once (The higher, the better).

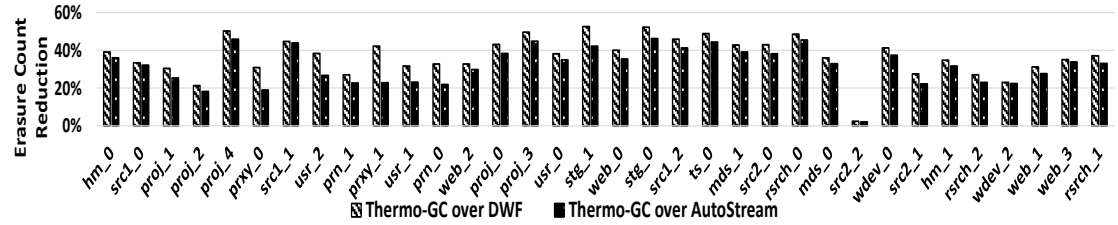


Figure 15: The erasure count reduction of Thermo-GC over DWF and AutoStream (The higher, the better).

sures. Therefore, Thermo-GC can increase the endurance of SSD by triggering less GC and erasure operations. To better visualize the erasure count reduction of Thermo-GC over DWF and AutoStream, we plotted WA reduction in Figure 15 except for traces wdev_1 and wdev_3 since they did not trigger enough GC. From the figure we can see that, although the reduction ratio various on different workloads, Thermo-GC is able to achieve less erasure counts in all situations over DWF and AutoStream without exception, which indicates better SSD endurance under Thermo-GC. One may notice Thermo-GC achieved very low erasure count reduction on trace src2_2. After looking into the trace src2_2, we found that src2_2 has very short update recency resulting in less valid pages in the GC candidate block, the average number of valid pages is 5.86 and 5.36 for DWF and AutoStream, respectively. Although Thermo-GC reduced the number of valid pages in a GC candidate block from above 5 to less than 1, Thermo-GC can not get much improvement from it because the valid page movements per GC are small.

2.6.3 Performance

The major advantage of Thermo-GC is that Thermo-GC minimizes the impact on user I/O by moving the data's lifetime identification from user requests processing path to GC, and clusters user requests as a free ride of a page's invalid process. In order to show this advantage of Thermo-GC, we collected the average I/O response times of DWF, AutoStream and Thermo-GC of all 36 MSRC traces. Figure 16 shows the normalized values of I/O response time. Both AutoStream and Thermo-GC's response times are normalized to DWF. As shown in Figure 16, Thermo-GC has shorter response time (less than 1) in most cases. On average, Thermo-GC reduced the I/O response time by 36.9% and 31.9% when compared to DWF and AutoStream, respectively. The major response time improvements come from the back-end bandwidth released by Thermo-GC's WA reduction. One may notice Thermo-GC has longer response time on traces src1_0 and proj_2 in addition to src2_2, wdev_1 and wdev_3 that Thermo-GC does not work well on as discussed previously. Traces src1_0 and proj_2 have the heaviest I/O models that exceed the processing power of the simulated SSD. Any additional processing can increase user I/O response time. We expect Thermo-GC has lower response time when the processing power of SSD is increased.

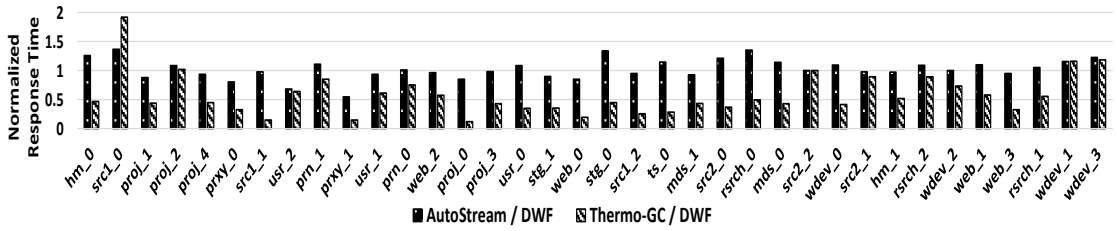


Figure 16: The normalized response time (The lower, the better).

2.6.4 Overhead Analysis

The resource analysis based on the configuration shown in Table 3. As mentioned in Section 1.2, Thermo-GC requires a time stamp per block to estimate the valid page's lifetime in it. In our current implementation, each time stamp takes 8 bytes, equals to the size of system time value. This takes 512KB DRAM for the 128GB SSD. Another 3 bits per block are needed to maintain grouping results, which takes additional 24KB DRAM. Therefore, the total DRAM requirement of Thermo-GC is 536KB. Since the measurement unit of AutoStream is 2MB as suggested in [17] and each unit requires 16B, the total memory requirement of AutoStream is 1MB. DWF does not require extra memory resource because DWF only separates pages by redirecting GC write requests to dedicated GC open blocks. As DWF, Thermo-GC is not noticeable by users, because Thermo-GC identifies pages' hotness during GC and take a free ride of the page invalid process. Thermo-GC takes less than 10 CPU cycles to separate a user write request. However, AutoStream needs to calculate the group number for each write request which takes around 550 CPU cycles.

2.7 Related Work

Extensive research has been carried out to reduce the negative impact of GC in NAND flash based SSDs [19, 20, 21, 22]. In this section, we discuss prior research that is closely related to Thermo-GC.

WA can be reduced by seperating hot and cold data. Chang et al. [8] adopt two-level LRU list to identify hot/cold write requests. Hsieh et al. [9] use multiple hash functions with a bloom filter to identify hot data. Park et al. [10] extend Hsieh's work by introducing multiple bloom filters to capture recency information in addition to frequency. Chiang et al. [7] present DAC to reorganize data in flash memory. DAC divides flash memory to multiple regions. Each region represents

different levels of hotness. Data are clustered into regions based on data's update frequency. Stoica and Ailamaki [23] propose a data placement algorithm to reduce GC overhead mainly based on update frequency. While the above research has made advances in separating hot and cold data, none of them takes advantage of GC procedure to identify data's lifetimes. Moreover, existing research identifies hot/cold pages in user I/O processing path that increases user I/O response time. On the contrary, Thermo-GC identifies a page's hotness during GC which does not impact user requests.

Recently, many research takes advantage of the multi-streamed SSD mechanism to reduce WA [24, 17, 25, 26]. Multi-Stream SSD [24] exposes physical blocks of SDD to user applications, and let programmers choose a block to write based on the data's life-cycles. AutoStream [17] uses the update frequency to identify hot pages and separates them to different groups. PCStream [25] uses program counter to classify write requests with different lifetimes into different blocks. FStream [26] separates journal and metadata requests of a file systems and maps these data to different blocks. However, AutoStream, PCStream and FStream require modifications in the operating system. In contrast, SSD with Thermo-GC is a plug and play device that entirely works inside SSD and does not require any modification in the operating system.

Other research uses greedy reclaiming policy [27, 3, 4, 5, 6] or increases over-provisioned space [28, 29, 30] to reduce WA. Nevertheless, none of them reduces the lifetime variance of pages in GC candidate block which is the root cause of WA.

2.8 Conclusion

In this paper, we proposed a new approach to minimizing WA and GC cost, namely Thermo-GC. This is the first paper, to the best of our knowledge, that

leverages the valid pages' lifetime information of GC candidate blocks to mitigate data migration caused by GCs. By using both valid pages' recency and frequency information during GC, we have successfully identified data lifetimes inside SSD with low overhead. Results show that Thermo-GC can reduce WA by 78% on average. More importantly, Thermo-GC reduces user requests response time because it frees up SSD's back-end bandwidth releasing by minimizing valid page movements during GC operations.

Acknowledgments

We would like to thank Dr. Qing Yang for his guidance and comments that helped greatly in improving the quality of the paper. We would also like to thank the anonymous reviewers for their valuable comments on this work. This research is supported in part by the NSF grants CCF-1439011 and CCF-1421823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. It is also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd.

List of References

[1] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvm," *ACM Trans. Storage*, vol. 13, no. 2, pp. 12:1–12:13, May 2017.

[2] G. Xu, M. Wang, and Y. Liu, "Swap-aware garbage collection algorithm for nand flash-based consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 1, pp. 60–65, February 2014.

[3] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system." in *USENIX*, 1995, pp. 155–164.

[4] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[5] L. Han, Y. Ryu, and K. Yim, "Cata: a garbage collection scheme for flash memory file systems," in *International Conference on Ubiquitous Intelligence and Computing*. Springer, 2006, pp. 103–112.

[6] B. Van Houdt, "Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data," *Performance Evaluation*, vol. 70, no. 10, pp. 692–703, 2013.

[7] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software: Practice and Experience*, vol. 29, no. 3, pp. 267–290, 1999.

[8] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 2002, pp. 187–196.

[9] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[10] D. Park and D. H. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, 2011, pp. 1–11.

[11] D. Shasha and T. Johnson, "2q: A low overhead high performance buffer management replacement algoritm," in *Proceedings of the Twentieth International Conference on Very Large Databases, Santiago, Chile*, 1994, pp. 439–450.

[12] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[13] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, no. 2003, 2003, pp. 115–130.

[14] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue {SSD} devices," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 49–66.

[15] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 10, 2008.

[16] B. Van Houdt, "On the necessity of hot and cold data identification to reduce the write amplification in flash-based ssds," *Performance Evaluation*, vol. 82, pp. 1–14, 2014.

[17] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Autostream: Automatic stream management for multi-streamed ssds," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 3:1–3:11.

[18] Micron. "Micron 3d nand flash memory." 2016. [Online]. Available: https://www.micron.com/~/media/documents/products/product-flyer/3d_nand_flyer.pdf

[19] M. Jung and J. Yoo, "Scheduling garbage collection opportunistically to reduce worst-case i/o performance in solid state disks," *Proceedings of IWSSPS*, 2009.

[20] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension." in *FAST*, vol. 12, 2012, pp. 10–10.

[21] J. Lee, Y. Kim, G. M. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 12–21.

[22] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 15–28.

[23] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 733–744, 2013.

[24] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 13–13.

[25] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, "Fully automatic stream management for multi-streamed ssds using program contexts," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 2019, pp. 295–308. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kim-taejin

[26] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J.-Y. Hwang, S. Cho, D. D. Lee, and J. Jeong, "Fstream: managing flash streams in the file system," in *16th USENIX Conference on File and Storage Technologies*, 2018, p. 257.

[27] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," in *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5. ACM, 1994, pp. 86–97.

[28] R. Lucchesi, "Ssd flash drives enter the enterprise," *Silverton Consulting. accessed on*, vol. 8, p. 2008, 2011.

[29] G. Drossel, "Methodologies for calculating ssd useable life," in *Proc. Storage Developer Conf*, 2009.

[30] Seagate. "Ssd over-provisioning and its benefits." 2019. [Online]. Available: https://www.seagate.com/tech-insights/ssd-over-provisioning-benefits-master-ti/

MANUSCRIPT 3

F/M-CIP: Implementing Flash Memory Cache Using Conservative Insertion and Promotion

Jing Yang[1], Qing Yang[2]

is published on the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: jyang@ele.uri.edu

[2]Distinguish Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: qyang@ele.uri.edu

Abstract

Flash memory SSD has emerged as a promising storage media and fits naturally as a cache between the system RAM and the disk due to its performance/cost characteristics. Managing such an SSD cache is challenging and traditional cache replacements do not work well because of SSDs asymmetric read/write performances and wearing issues. This paper presents a new cache replacement algorithm referred to as F/M-CIP that accelerates disk I/O greatly. The idea is dividing the traditional LRU list into 4 parts: candidate-list, SSD-list, RAM-list and eviction-buffer-list. Upon a cache miss, the metadata of the missed block is conservatively inserted into the candidate-list but the data itself is not cached. The block in the candidate-list is then conservatively promoted to the RAM-list upon the k-th miss. At the bottom of the RAM-list, the eviction-buffer accumulates LRU blocks to be written into the SSD cache in batches to exploit the internal parallelism of SSD. The SSD-list is managed using a combination of recency and frequency replacement policies by means of conservative promotion upon hits. To quantitatively evaluate the performance of F/M-CIP, a prototype has been built on Linux kernel at the generic block layer. Experimental results on standard benchmarks and real world traces have shown that F/M-CIP accelerates disk I/O performance up to an order of magnitude compared to the traditional hard disk storage and up to a factor of 3 compared to the traditional SSD cache algorithm in terms of application execution time. Furthermore, F/M-CIP substantially reduces write operations to the SSD implying prolonged durability.

3.1 Introduction

Recent developments of flash memory based SSD (solid state disk) have been very promising with rapid increase in capacity and decrease in cost. Because SSD is a semi-conductor device, it provides us with great advantages in terms of high-speed random reads, low power consumption, compact size, and shock resistance. The performance/cost characteristics of the SSD make it a perfect fit for a cache layer between the system RAM and the disk storage[1, 2, 3]. However, traditional cache management algorithms such as LRU and its variants[4, 5, 6, 7] do not work well for SSD cache because of its write amplifications, wearing, and garbage collections.

To quantitatively observe how SSD performs as a cache, we have carried out extensive experiments to measure the I/O performance of one state-of-the-art high end PCIe SSD. We measured the I/O throughputs of a 320GB PCIe SSD by varying the allowed address space of accessed data sets as a percentage of the total SSD capacity. On the allowed address space, we repeatedly perform I/O operations on the SSD up to about 2 terabytes of data being written. For all I/O operations during the first 320GB, i.e. before the SSD is filled, the I/O performance achieved 500MB/s which is manufacturer announced performance number. However, immediately after 320GB, I/O performance drops rapidly for allowed address space of 50%, 70%, 90%, and 100% of the SSD capacity. The higher the percentage of allowed address space, the sharper the bandwidth drops. This write cliff happens on all SSDs we have experimented. In our experiments we also noticed that mingled reads and writes have significant impact on performance. Similar results were obtained for all other SSDs that we have experimented. Based on our experiments, several observations are in order.

Writes on flash memory SSD are costly and have significantly negative im-

pact on performance. This asymmetric read/write performance characteristic of SSD does not exist in traditional RAM caches. Performance of traditional cache increases as the cache size increases because cache hit ratio is a monotonically increasing function of the cache size. This is no longer the case for flash memory SSD cache. Therefore minimizing writes is important.

Traditional cache management algorithms such as LRU and its variants allocate space for most recently referenced blocks in the cache. Some of these allocated cache blocks may never get referenced after they are allocated in the cache such as scan accesses. Others may get a few cache hits but the benefit of the few hits may not compensate the cost of SSD writes due to repeated hits and misses such as thrash accesses. Researchers have long realized the write costs of SSD cache and have come up with various solutions such as separated read and write caches[1], selective cache[2], and compressed cache[8] to list a few.

Intermixing reads and writes randomly hurts flash memory SSD performance. Each plane of a typical SSD has one cache register and one data register as data buffers at the end of the plane. These two registers are connected in a serial fashion allowing pipelined operations for consecutive reads or consecutive writes on the same plane. For example, in Micron's flash memory, write pipeline happens when write data flows from the bus to the cache register as the first stage. At the second stage, the cache register passes the data to the data register after completely receiving the data to be written. Then write operation starts by moving data from the data register to flash memory cells as the final stage during which data transfer of the next write data from the bus to the cache register is done concurrently. Such pipeline operations will stall when a write is immediately followed by a read because data now flow in two different directions. Therefore, a good SSD cache design should minimize intermixes of random reads and writes as much as possible.

The last observation in our experiments is the inherent parallelism of I/O operations inside flash memory SSDs. The parallelism happens among multiple packages, chips, planes, and channels. This is particularly true for multiple write operations. Because writes are not in place but on available clean pages irrespective of the addresses of written data, a typical SSD controller is intelligent enough to find clean pages that can be written in parallel if there are multiple write requests pending. This is quite different from read operations that have to get data from where they were previously stored limiting the parallelism by the address distribution of pending reads. The parallel operation of I/Os inside SSD can greatly improve performance as observed by our experiments as well as other researchers previously[9]. Therefore, writing SSD in large batches can fully exploit its internal parallelism.

Motivated by our experimental observations, we propose a new cache management algorithm that makes the best use of the physical properties of flash memory SSD. The new cache algorithm is referred to as F/M-CIP (Flash Memory Cache with Conservative Insertion and Promotion). F/M-CIP accelerates I/O performance greatly when using flash memory SSD as a cache. The idea of F/M-CIP is managing a two level cache hierarchy consisting of a small RAM buffer on top of the SSD cache by maintaining a CIP-list that is divided into 4 parts: candidate-list, SSD-list, RAM-list and eviction-buffer-list. Upon a cache miss, the metadata of the missed page is conservatively inserted into the candidate-list but the data itself is not cached in the SSD. The page in the candidate-list is then conservatively promoted to the RAM-list upon the k-th miss. In this way, F/M-CIP effectively filters out one-time access I/Os as well as large and sequential I/Os with no write to the SSD cache. Not only does it minimize writes to the SSD cache, but also provide good I/O performance because disks perform well for large sequential ac-

cesses since the one time seek delay is amortized by the high throughput resulting from the large sequential data transfer. On the other hand, while the data is not cached upon the first miss, it will not result in significant performance penalty because the page will be cached at high-level caches such as page cache or buffer cache.

At the bottom of the RAM-list, we maintain an eviction-buffer that accumulates LRU pages to be written into the SSD cache in large batches containing 32 to 128 SSD pages. Destaging large batches of data from the RAM buffer to the SSD cache has several performance benefits. First of all, large batch writes provide the SSD controller with more opportunities to maximize internal parallelism to speedup write operations. Secondly, for the same amount of writes to the SSD, writing in batches creates large interval time between two subsequent writes. The effect is minimizing the chance of random intermixes of reads and writes that slow down I/O operations. The candidate-list and the RAM list work together to determine what data should be cached. Once a page is eligible for caching, it goes to the RAM cache first and is eventually destaged to the SSD cache in batches to maximize parallelism and pipelining performance in the SSD. The SSD-list manages cached data in the SSD using a combination of recency and frequency replacement policies by means of conservative promotion upon hits.

To quantitatively evaluate the performance of F/M-CIP, a prototype has been built on Linux kernel as a device driver at the generic block layer. The device driver uses a small partition of the system RAM as the RAM buffer on top of an SSD and implements the F/M-CIP functions to manage the two-level disk cache. Standard benchmarks and I/O traces were used to drive the prototype measuring I/O performance in comparison to existing disk I/O systems with an SSD cache using traditional cache management algorithms or without a cache. Experimental

results on standard benchmarks and traces have shown that F/M-CIP accelerates disk I/O performance by orders of magnitude compared to traditional hard disk storage and up to a factor of 9 compared to traditional SSD cache algorithm in terms of application execution time. Furthermore, F/M-CIP substantially reduces write operations to the SSD implying prolonged durability.

The paper is organized as follows. Next section presents the F/M-CIP architecture in details. Section 3.3 gives the design and implementation of the prototype F/M-CIP. Experimental settings and workload characteristics are presented in Section 3.4 followed by results and discussions in Section 3.5. We discuss related work in Section 3.6 and conclude the paper in Section 3.7.

3.2 F/M-CIP Architecture

The key to the F/M-CIP cache replacement algorithm is to maintain a CIP-List as shown in Figure 17. CIP-List is similar to the tag array of the CPU cache and contains the metadata of cached pages such as pointers, status, and LBAs (logic block address) of cached pages. We use "page" throughout this paper to refer to the basic data unit in our cache instead of "block" to avoid possible terminology confusion. Each node in the CIP-list needs less than 70 bytes for the necessary metadata resulting in less than 1.7% space overhead for page size of 4KB. We divide the CIP-List into four sub-lists: RAM-list, eviction-list, SSD-list, and candidate-list. Let LR, LS, and LC be the length of RAM-list, SSD-list, and candidate-list, respectively. Let B be the size of eviction buffer in terms of number of pages. Then the size of the RAM buffer that F/M-CIP manages is LR + B pages, the size of the SSD cache that F/M-CIP manages is LS pages, and there are LC candidate pages to be considered for eligibility for caching.

The RAM list is the traditional LRU list with the most recent referenced page on the top of the list and the least recently used page at the bottom of the

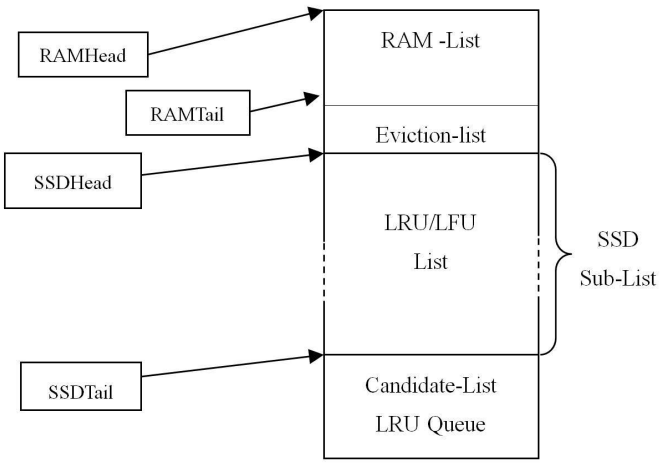


Figure 17: Block diagram of the CIP-List
- RAM-list managing the RAM cache
- SSD-list managing the SSD with block allocation and replacement
- Eviction-list accumulating LRU pages to be written in SSD
- Candidate-list filtering out sweep accesses to sequential data

list. A newly referenced page that makes to the RAM buffer is inserted at the top of the list. Upon a hit in the RAM buffer, the referenced page is deleted from its current position in the RAM-list and re-inserted on the top of the list. When replacement from the RAM buffer is necessary, the page at the bottom of the RAM-list is removed. If the removed page is not in SSD-Consistent (to be discussed shortly) state, it is inserted into the eviction-list and the page data is stored in the eviction buffer. Whenever the eviction buffer is full with B pages, SSD writes are triggered to write all B pages into the SSD in one batch and the eviction buffer is emptied. As mentioned in the introduction, such batch writes maximize the internal parallelism of SSD because the SSD controller will schedule the writes on clean pages across multiple planes irrespective of the LBAs of data to be written. The batch writes also minimize the chance of intermix of random reads and writes giving rise to better pipelining performance inside the SSD.

The SSD-list manages cached pages in the SSD cache using a frequency-based replacement. Pages at the top of the SSD-list are frequently referenced pages and

pages at the bottom are least frequently referenced pages. Every time there is a read hit in the SSD, the referenced page is conservatively promoted by moving it one position up in the SSD-list. A new page entering the SSD is initially inserted at the top of the list to give it a chance to show its importance. If it gets re-referenced often, it will stay on the top or the upper portion of the SSD-list. If it is not frequently reference page, it will get demoted down the list and give way to other more frequently referenced pages. When replacement from the SSD cache is necessary, the pages at the bottom of the SSD-list will be evicted from the cache. The SSD-list accurately reflects the frequency of page references once pages enter the SSD cache. Since SSD cache tends to be very large in the range of hundreds of GB, our experiences have shown that frequency-based replacement provides good cache performance. For workloads that favor recency replacement, the RAM buffer can catch most of recency references as will be evidenced shortly.

The candidate-list is a LRU queue that keeps track of LC missed pages in time order. Data corresponding to the pages in the candidate-list are not cached in the F/M-CIP cache. Instead only the LBAs of pages are kept in the candidate-list. Whenever a page in the candidate-list is referenced again, it is eligible for caching. The pages metadata is promoted to the RAM-list and its data is cached in the RAM buffer. Therefore, a missed page is given an opportunity to compete for caching for the entire time it spends in the candidate-list. The length of this candidate-list is an important parameter that impacts on the cache performance. For high memory efficiency and low overheads, we would like to have a short candidate-list. If the candidate-list is too short, on the other hands, many potential hot data pages may never get a chance to make into the cache if they are in between sweep accesses to large sequential data that is longer than the candidate-list. Such large sequential data will flush out those random data with potentially good locality before they

get referenced again. The purpose of the candidate-list is to filter out those one-time access pages (cold pages) and those large sequential references. Caching such cold pages or large sequential data may not provide much performance benefit but may hurt cache performance because of excessive writes to SSD. In addition, it is known that hard disk drive performs fairly well for large sequential data with I/O throughput comparable to that of SSD if the data size is large enough. With proper determination of the length of the candidate-list, F/M-CIP can effectively filter out cold and sequential data and be scan resistant. Pages in a scan access sequence will not make to the SSD-list if they are not re-referenced while in the candidate-list and therefore are replaced from the candidate-list before they can be cached in the SSD.

The candidate-list described above allows a page to be cached upon the second miss. For some workloads, it may be beneficial to allow a page to be cached only after the k-th miss. This is because a notable fraction of pages got referenced for only a few times while other pages with good locality got referenced tens even hundreds of times[2]. We will discuss how to implement candidate-list allowing a page to be cached only after k-th miss in the next section. Each cached page has two status bits: SSD-Consistent and HDD-Consistent. The SSD-Consistent bit indicates whether the RAM copy of the page is consistent with that in the SSD cache and the HDD-Consistent bit indicates whether a cached copy in the RAM buffer or the SSD cache has been changed since it was loaded to the cache. A page can be in Not-SSD-Consistent in two cases. First, the page has been changed since the last time it was copied from SSD to RAM. Second, SSD does not have the page yet and it needs to be written to SSD when evicted from the RAM buffer. Replacing an SSD-Consistent page from the RAM buffer to the SSD does not require SSD write whereas replacing a HDD-Consistent page from the cache does

not need HDD operation. In the following paragraphs, we will discuss how the F/M-CIP cache works.

Cache Hits:

(i) If the referenced page, p, is in the RAM-list of the CIP-List, return data and p is placed on the top of the RAM-list. If the reference is a write, change the status of the page to Not-SSD-Consistent and Not-HDD-Consistent.

(ii) Upon a read reference to page p that is in the SSD-list but not in the RAM-list, p is promoted by one position up in the SSD-list. If it is on the top of SSD-list it is inserted to the top of the RAM-list with SSD-Consistent state and the data is copied into the RAM buffer. If the size of the RAM-list is LR at time of the insertion, the page at the bottom of the RAM-list needs to be evicted. If the evicted page is in SSD-Consistent state, drop the page to make room for the newly inserted page. Otherwise, the evicted data page is moved from the RAM buffer to the eviction buffer to make room in the RAM buffer for the newly inserted page. If the number of pages in the eviction buffer reaches B, perform SSD_Write.

(iii) Upon a write reference to page p that is in the SSD-list, p is promoted to the top of the RAM-list with state being set to Not-SSD-Consistent and Not-HDD-Consistent after the page is written in the RAM buffer. If the size of the RAM-list is LR at time of the write, the page at the bottom of the RAM-list is evicted. If the evicted page is Not-SSD-Consistent, it is demoted to the eviction-list and its corresponding data page is moved from the RAM buffer to the eviction buffer. If the number of pages in the eviction buffer reaches B, perform SSD_Write.

Cache Misses:

Check if page p is already in the Candidate-list.

- If p is not in the Candidate-list, add the LBA of p to the top of the candidate-list. If the length of the candidate-list at the time is greater than LC, delete a node at the bottom of the candidate-list.

- If it is in the Candidate-list, load p into the RAM buffer. If it is a read miss, set the state of p to Not-SSD-Consistent and HDD-Consistent. If it is a write miss, set the state of p to Not-SSD-Consistent and Not-HDD-Consistent. Insert the LBA and the state of p to the top of the RAM-list. If the RAM buffer is full at the time of the miss, RAM buffer replacement is done in the same way as described in Cache Hits-ii above.

SSD_Write:

(i) If SSD is full, i.e. SSD-list size equals LS, destage the bottom B pages in the SSD-list to the HDD. Only those pages in Not-HDD-Consistent state need to be read from the SSD and written to HDD.

(ii) Perform SSD writes by moving all data pages in the eviction buffer to the SSD. Clear the eviction buffer.

3.3 Prototype Design and Implementation

A working prototype has been developed under the Linux kernel as a generic block device driver in parallel to LVM (logic volume manager) or software RAID drivers. The F/M driver intercepts all block I/Os issued from upper layer OS and applications. The intercepted block I/Os are then processed using the F/M-CIP cache algorithm. It uses a small raw partition of the system RAM to implement the RAM buffer, eviction buffer, and metadata structure and calls upon the physical device drivers such as SATA and PCIe to accomplish all possible I/O operations issued from the upper layers.

The CIP-list is implemented as a doubly linked list with each node containing metadata information for each cached page including its LBA, state, and necessary pointers.

The candidate-list is implemented in two different structures: queue-based and table-based. In the queue-based implementation, LBAs of newly missed pages are appended at the tail of the queue and removal is done at the head of the queue to keep the queue length a constant. Queue-based implementation would become more complicated if we were to allow caching after the k-th miss.

In the table-based candidate-list implementation, we keep a timer and a counter in each table entry that is hashed by LBA. The counter determines when a missed page is eligible for caching. It is incremented each time a miss occurs and when it exceeds a predetermined threshold value, say k-th miss for threshold value of k, the corresponding page is cached in the RAM buffer. The timer is used to determine when the counter should be reset to 0. Although the table-based implementation allows more flexible cache filtering with the counter and consumes less memory space than queue-based, it has several shortcomings. First of all, table-based candidate-list structure is not able to keep strict time order or LRU order of the candidate LBAs. As a result, it cannot provide a fair chance to all candidate LBAs to compete for caching between two subsequent timer resets. Secondly, hash collision may result in inaccurate miss counts giving rise to incorrect cache decision of a page that is missed first time but its LBA is hashed to an entry with miss count exceeding the threshold value due to another hot page. Nevertheless, the table-based candidate-list does provide fairly good approximation of the desired filtering effect of F/M-CIP algorithm as evidenced in our experimental result section. Therefore, both queue-based and table-based candidate-lists are viable options that a designer can use to exercise trade-offs in practice depending on

application workloads and memory pressure of the server system. Our prototype implemented both structures for the candidate-list.

3.4 Experimental Setup and Workload Characteristics

3.4.1 Experimental Settings

In order to quantitatively evaluate the performance of our F/M-CIP cache architecture and compare it with existing architectures, we have installed the F/M prototype driver on 6 Dells PowerEdge R310 rack servers running Linux in our lab. Each rack server has an Intel Xeon X3460 processor with 2.80 GHz and 8M Cache. The system RAM of each server is 8GB with 1333MHz and Dual Ranked UDIMM. The hard disk drive used in each server is 160GB 7.2K RPM SATA drive. In addition to the F/M-CIP cache driver, we have also installed an existing and well-known cache software called Flashcache[10] that uses the traditional cache replacement algorithm to manage SSD as a cache for HDD. We intended to compare our F/M-CIP cache with Flashcache as well as hard drive only system as a baseline. Throughout all our experiments, we set the page size to be 4KB and eviction buffer size to be 64 pages.

Since our experiments run on multiple GB of data, each run takes hours to finish. We therefore carried out our experiments in parallel on the 6 servers. We tried to use all the SSDs that are available in our lab although they are different types, 2 PCIe SSD cards and 4 SATA SSDs, as shown in Table 4. To make sure that our performance comparison is fair, we ensure that each benchmark will run for three times for the three different storage architectures, F/M-CIP, Flashcache, and hard disk drive (HDD) only, respectively, using exactly the same hardware settings. Readers may notice discrepancies among the performance results across different benchmarks due to different SSDs used. However, we are only interested in the relative performances of the three different storage architectures under the

same hardware environment.

3.4.2 Workloads

The workloads we used in our experiments consist of 4 standard benchmarks and 2 real world I/O traces as listed in Table 4. These benchmarks cover a wide spectrum of applications that require high I/O performance.

OLTP benchmarks: SysBench is a modular, cross-platform, and multi-threaded benchmark for systems under intensive I/O loads. In our experiments, Sysbench was run against MySQL database with table size of 4,000,000, max requests of 500,000, and 16 threads.

File System Benchmarks: One of the PostMark applications is the simulation of email servers. In order to provide complete reproducibility, PostMark is composed by three phases. The first phase generates a pool of files. The second phase includes file generation, file deletion, file reading, and file appending. In the last phase, all created files in the pool are deleted. In our experiments, we set PostMark workload to include 75,000 files and to perform 200,000 transactions. File sizes are set to 4KB to 64KB.

Financial Analysis: SAS environment was provided by SAS Institute and run a typical transaction cost estimate process[11] on financial history data that are publicly available. This transaction cost analysis program is both CPU and I/O intensive to process a large amount of data using a complex statistic analysis tool. We ran this analysis program in the SAS environment on 14.6GB of historical data to measure the total execution time.

Biomedical Applications: Another class of important applications is biomedical engineering that requires high performance and data intensive computation. A typical example is genome and EST sequence assemblers that are widely used in biomedical research and drug development applications. One of such open

source programs is Mira, a multi-pass DNA sequence data assembler/mapper for whole genome and EST projects. The dataset we used for Mira is downloaded from NCBIs sequence read archive. The total size of the data set is 3.6GB. Such assembly process generates explosive amount of data and requires a large amount of system RAM. When the RAM size is limited, it puts a burden to disk storage system used as swap space.

SPC-1 Traces: SPC-1 traces are from Storage Performance Council that consists of a single workload performing the typical functions of business critical applications. Those applications are characterized by predominately random I/O operations and require both queries and update operations. Examples of those types of applications include OLTP, database operations, and mail server implementations. SPC Financial is a well-known block level disk I/O traces taken from OLTP applications running at two large financial institutions[12], Finacial-1 and Financial-2. These two sets of traces were replayed on our Linux virtual machines to generate I/O operations to the underlying storage system.

Mixed Workloads: Server consolidation and cloud computing often use virtualizations. In such environment, multiple virtual machines that carry out different applications are installed and run on a single physical server host. To evaluate how such mixed workloads affect cache performance of F/M-CIP, we have set up multiple VMs that run different benchmarks on one host machine. In such a system, multiple virtual machines running benchmarks generate I/Os to the host hypervisor where our storage drivers are installed. The storage system receives aggregated I/O requests from all virtual machines. This is an interesting test to the performance of various storage architectures and is close to real world applications.

Table 4 summarizes the 6 different experimental runs using the standard benchmarks and traces. The first experiment had two virtual machines with one

Table 4: Experimental characteristic

Experiments	RAM		SSD			I/O sizes (GB)		
	System (VMs excluded)	CIP RAM buffer size	Manufacture	Interface	SSD cache size	Read	Write	Totoal
Sysbench OLTP & PostMark	700MB	256MB	OCZ(60G,RAID 1)	PCI-E	2048MB	17	15	32
DB Cluster	600MB	256MB	Crucial(64G)	SATA	2048MB	58	27	85
SAS	2048MB	1024MB	Crucial(64G)	SATA	4096MB	187	24	211
Mira	700MB	256MB	FusionIO(80G)	PCI-E	4096MB	33.5	11.5	45.0
SPC-1 Financial 1	600MB	256MB	OCZ(120G)	SATA	2048MB	1.1	29	31.1
SPC-1 Financial 2	600MB	256MB	Intel510(120GB)	SATA	1536MB	2.0	5	7.0

running OLTP benchmark (SysBench) while the other running file system benchmark (PostMark). The I/O activities seen from the hypervisor at the server host are mixed workloads. The second experiment had a database cluster (DB cluster) of 6 VMs all running SysBench. The remaining 4 experiments ran each individual benchmarks in isolation as shown in Table 4. The total amount of I/Os in each experiment ranges from 7GB to 211GB as seen by our F/M-CIP driver at the generic block layer as shown in the last column of Table 4. That is, these I/Os have passed the system buffer cache and have arrived at the storage system controlled by F/M-CIP. In order to generate enough I/O activities to evaluate caching and replacement activities for the given data set size of each benchmark, we limit the system RAM size in each experiment to be substantially smaller than the data set size and restrict the amount of SSD space for our cache.

3.5 Results and Discussions

In this section, we report and discuss our experimental results. Hit ratio has been traditionally used to evaluate cache performance. However, in our environment, hit ratio alone is not sufficient to give enough performance insights of different cache designs. F/M-CIP uses a small partition of the system RAM as a buffer to manage the two level cache hierarchy whereas Flashcache does not take much system RAM except for meta data. For example, if the system RAM is 1GB and F/M-CIP uses 200MB for the RAM buffer, then the cache managed by F/M-CIP is 200MB larger than that by Flashcache but it will receive more I/Os than

Table 5: Total running time of four experimental runs

Storages benchmarks	S&P Mix	DB Cluster	Mira	SAS
F/M-CIP Qbased	3,020.69	12,415.54	6,537.60	3,346.80
F/M-CIP Tbased	3,912.55	13,819.44	6,831.00	4,746.00
Flashcache	5,466.91	36,286.61	7,857.00	5,604.00
HDD	193,89.89	127,026.23	10,591.80	3,739.80

Flashcache because the system buffer cache of F/M-CIP is reduced by 200MB. For example, when we run Financial-1 of SPC-1, Flashcache received 29.8GB whereas F/M-CIP received 31.1GB because of reduced system buffer cache above the block layer. Since both F/M-CIP and Flashcache use exactly the same hardware and the same amount of system RAM, the execution time of a benchmark or query response time from user point of view will provide a fair performance metric for our performance comparison purpose.

Table 5 shows the total running times of the first four experiments in terms of seconds. We have run both queue-based (Qbased) F/M-CIP and table-based implementations as described in Section 3.3. For table-based candidate-list (Tbased) we set the counter to 2 and timer to 600s based on our experimental observations. As shown in this table, F/M-CIP dramatically reduced the total execution times of DB Cluster and OLTP/Filesystem mix experiments compared to hard disk storage system with no cache. To clearly show the performance improvements, we drew the speedup bar graphs as shown in Figures 18(a) and (b) compared to hard disk drive and Flashcache, respectively. For queue-based F/M-CIP, the speedup over hard disk storage ranges from 12% to an order of magnitude and the speedup over Flashcache[10] ranges from 20% to about a factor of 3. For Mira and SAS, the performance improvements are limited in the range of 12% and 67% with queue-based candidate-list. With table-based candidate-list, we noticed that it performs about 21% worse than hard disk storage. This is also the case for Flashcache that

Table 6: Average response time of individual queries for database benchmarks

Respones Time(ms)	DB Cluster	S&P Mix
F/M-CIP Qbased	442.18	96.63
F/M-CIP Tbased	397.22	125.18
Flashcache	1,277.47	174.90
HDD	4,063.38	620.44

shows about 50% slow down compared to hard disk only storage system. To understand why this is happening, we took a close look at the I/O behavior of the SAS program. We noticed that there is a large fraction of I/O operations that are one time and sequential accesses. As mentioned previously, hard disk storage performs well for large sequential accesses. If we try to cache these large sequential accesses in the SSD, the overhead of allocating space in SSD and replacing them from SSD will slow down I/Os with no cache benefits. However, F/M-CIP using queue-based candidate-list can effectively filter out such accesses and provide performance improvement of 12% over hard disk only and 67% over Flashcache as shown in Figure 18.

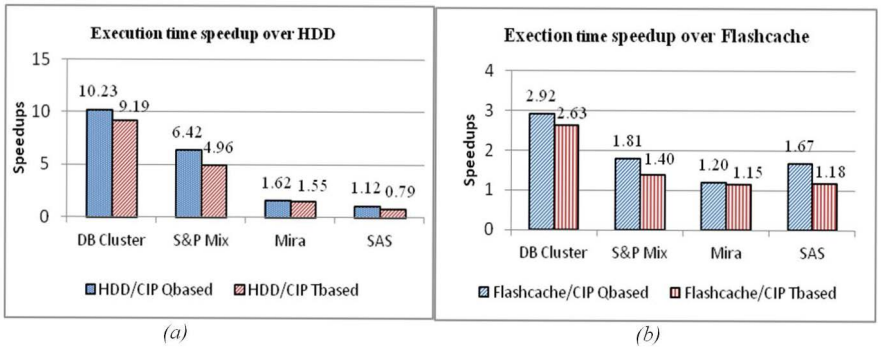


Figure 18: F/M-CIP speedups in terms of total execution time of the benchmarks.
(a) Speedup over hard disk drive.
(b) Speedup over traditional cache management algorithm, Flashcache.

In addition to the total execution times, some database benchmarks also measure the query response time such as SysBench. Table 6 lists the measured average query response times of two database benchmarks in terms of milliseconds. It is clearly shown in Table 6 that our F/M-CIP accelerates query response time greatly

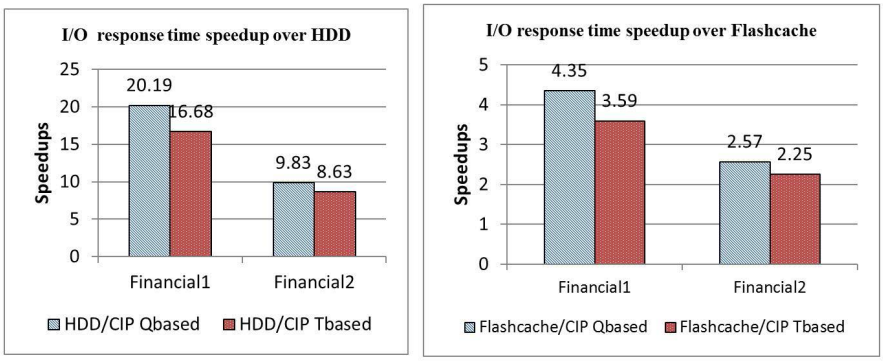
over both hard disk only and Flashcache. It is interesting to note the substantial performance improvement of F/M-CIP over both hard disk only and Flashcache for Database Cluster. It provides a speedup of 10 times over hard disk only storage and a speedup of 3 times over Flashcache. This remarkable performance improvements clearly show the superb efficiency of our new F/M driver. SysBench benchmark is a type of stress test that generates high intensive I/Os with mixed workloads. For such high intensity I/O workloads with a variety of request types, F/M-CIP clearly shows its advantage in effective cache management. It filters out cold and large sequential accesses and cache random accesses with good locality.

Table 7 lists the results of SPC-1 traces. We measured both the average response time of individual I/O operations and the total running time of the traces. It is clearly shown that F/M-CIP driver substantially reduced both the average I/O time and total running time for both traces: Financial-1 and Financial-2. While both F/M-CIP and Flashcache provide substantial speedups over the hard disk only storage system, F/M-CIP shows greater cache advantages than Flashcache indicating the effectiveness of the new cache algorithm. Figures 19(a) and (b) show the speedups of our F/M-CIP over hard disk only and Flashcache driver, respectively. We observed a factor over 20 speedup of queue-based F/M-CIP over hard disk only storage system and a factor of 4.35 speedup over Flashcache driver for Financial-1 traces. For Financial-2 traces that have less locality than Financial-1, we still observed speedup of 9.83 and 2.25 times over hard disk only and Flashcache, respectively. With table-based candidate-list, the speedup is slightly lower but still substantial compared to both hard disk only and Flashcache systems.

In addition to the superb performance advantages, our new F/M-CIP driver substantially reduced the total number of writes to SSD. As we know, flash memory SSD has limited life time constrained by the total number of erasure operations.

Table 7: Average I/O response times and total running times of SPC-1 I/O traces

Storages	Financial-1		Financial-2	
	I/O Resp. (ms)	Running Time(s)	I/O Resp. (ms)	Running Time(s)
F/M-CIP Qbased	3.620	2,418	2.027	941
F/M-CIP Tbased	4.383	2,926	2.311	1,072
Flashcache	15.744	10,455	5.206	2,403
HDD	73.093	48,735	19.933	9,021



(a) (b)

Figure 19: F/M-CIP speedups in terms of average I/O response time and total running time of SPC-1 I/O traces.
(a) Speedup over hard disk drive.
(b) Speedup over traditional cache management algorithm, Flashcache.

A typical SLC (single level cell) flash memory can survive about 100k erasure cycles while an MLC (multi level cells) can survive a few thousands erasure cycles. Therefore, reducing writes to an SSD prolongs the life time of the SSD, which is very important for enterprise applications. In our experiments, we measured the total number of write operations performed to the SSD cache for each benchmark for both F/M-CIP driver and Flashcache driver. Table 8 shows the write reduction ratio, the total number of SSD writes of Flashcache divided by the total number of SSD writes of F/M-CIP. As shown in the table, F/M-CIP reduced the total number of SSD writes greatly compared to Flashcache. The reduction ranges from 90% to a factor of 11 depending on applications. We noticed that in most cases queue-based candidate-list filters out cold data and large sequential accesses better than table-

Table 8: Write reduction ratio: Writes2SSD_FC/Writes2SSD_CIP

Config Benchmarks	DB Cluster	S&P Mix	Miro	SAS	SPC-1	
					F-1	F-2
F/M-CIP Qbased	11.3	9.3	3.0	3.3	4.8	4.6
F/M-CIP Tbased	5.0	2.3	1.9	3.7	4.8	6.1

based candidate-list. This is mainly because of aliasing problem of table-based that could cache data inaccurately. However, in some cases where applications show very good data locality, table-based candidate-list shows a little more write reduction than queue-based does. In this applications, table-based candidate-list has slightly less writes than queue-based candidate-list since a data page is cached only after the page has missed two times.

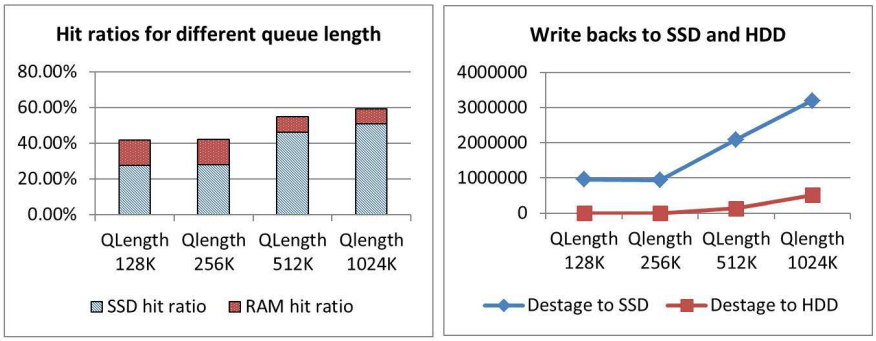


Figure 20: Effects of candidate-list length on cache hit ratios, SSD writes, and SSD to HDD destages.
(a) Cache hit ratios corresponding to different candidate-list length.
(b) Number of write I/Os to SSD and number of destages from SSD to HDD

As mentioned in the last section, the length of candidate-list is an important parameter to cache performance that needs to be determined carefully. To quantitatively evaluate performance impacts of this parameter on cache performance, we ran the Sysbench and PostMark mix experiment and measured hit ratios as a function of the candidate-list lengths as shown in Figure 20(a). It is shown in this figure that the hit ratio increases as the candidate-list increases, which is intuitive and easily understandable. The longer this candidate-list is, the more time each

page has for a chance to be cached in the F/M-CIP cache. As a result, cache hit ratio increases. However, large candidate-list consumes more system RAM resulting in high space overhead. More importantly, increasing the candidate-list length also contributes more writes to SSD and destaging operations from SSD to HDD as shown in Figure 20(b). Such increase may adversely impact the overall system performance because of the high cost of SSD writes and time consuming destaging operations each of which requires a read from SSD and a write to HDD.

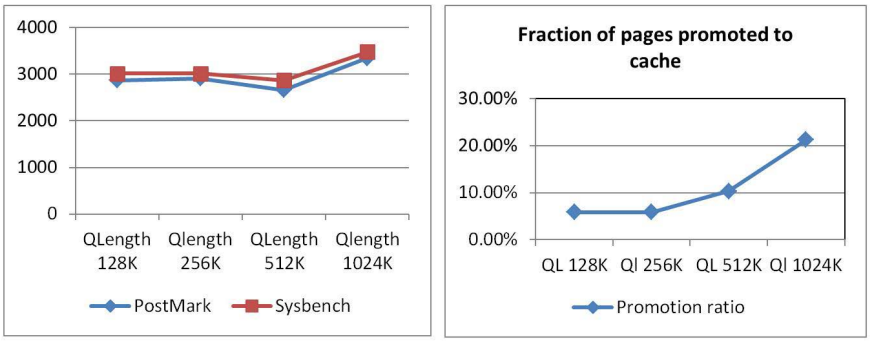


Figure 21: Performance impacts of the length of candidate-list.
(a) benchmark execution time for different candidate-list length.
(b) proportion of pages promoted to the cache for different candidate-list lengths.

Figure 21(a) shows the benchmark execution times of Sysbench and Postmark as a function of the candidate-list length. This results seem to be counter intuitive at the first look because of the longer execution time of higher cache hit ratio than that of lower hit ratio. This performance abnormality demonstrated again our analysis discussed in the introduction: design of flash memory cache is quite different from traditional RAM cache because of its asymmetric physical property and high cost of writes. The function of our candidate-list is to filter out cold and large sequential accesses and cache those data that have short re-reference interval times. When the candidate-list is too long, many pages that have long re-reference intervals get promoted to the cache. Particularly when the candidate-list contains the number of pages larger than that the cache can hold, data thrashing between the cache and hard disk happens giving rise to longer execution time. This fact is

clearly demonstrated in Figure 21(a) when the candidate-list contains 1024K nodes while the total cache size is only half of the pages (cache size=2GB=512K pages < candidate-list=1024 pages = 1024*4K=4GB). Figure 21(b) shows the proportion of pages promoted to the cache from the candidate-list. With candidate-list being 1024K nodes, over 20% of all accessed pages are cached while about 10% of data are hot pages that should be cached. Therefore, filtering out those pages with large re-reference interval is very important to SSD cache performance because of the cost of writes and data thrashing between SSD and hard disk drive. F/M-CIP cache with properly chosen candidate-list can effectively filter out thrash accesses and large sequential accesses.

3.6 Related work

LRU (least recently used) stack has been successfully used in traditional cache designs for cache replacement. There have been several interesting variations in terms of insertion, promotion, and demotion of the logic position of a referenced line in the LRU stack[4, 5, 6]. These variations make adaptive decisions based on cache access behavior at run time and have shown performance improvements over traditional LRU. Gao and Wilkersons DSB (Dueling Segmented LRU replacement algorithm with adaptive Bypassing)[7] employs a bypassing logic in each set to dynamically decide whether to bypass a missed line (competing line) or cache the missed line by evicting the victim LRU line. There are many significant differences between DSB and F/M-CIP in their fundamental techniques in addition to the fact that they address completely different level caches: CPU cache vs. storage cache. Bypassing in F/M-CIP is not probabilistic rather a miss on the first access of a page will be bypassed for sure. A page can make to the cache only on the second or a later access depending on the predetermined threshold value. F/M-CIP focuses on minimizing writes to SSD by giving high priority to cached pages in SSD to

stay in the cache until a page in the candidate-list turns hotter than an LRU page in the cache when working set shifts.

Jaleel et al[6] proposed a cache replacement using Re-reference Interval Prediction (RRIP). An M bit counter is used to keep track of re-reference intervals of each cache line with 0 being the near-immediate re-reference and 2M-1 being the distance re-reference. The goal is to keep cache lines with short re-reference interval and replace lines with distance interval. Their Static RRIP (SRRIP) is scan-resistant and their Dynamic RRIP (DRRIP) is both scan-resistant and thrash-resistant. Both replacements provide performance improvements over traditional LRU and LFU algorithm. The conservative insertion and promotion of the F/M-CIP algorithm share some similar ideas to RRIP that predicts a long re-reference interval value for a missed page and gradually increase the value upon hits (frequency priority). However, for set-associative CPU cache design, the degree of associativity rarely goes over a hundred while for storage cache that is usually fully associative, the number of pages for replacement management can easily go over millions. As a result, the space overhead for keeping the counters in a storage cache is much higher than the CPU cache. More importantly, updating these counters across millions of pages for each reference is prohibitively time consuming rendering the counter for predicting re-reference interval value impractical for storage cache that is large and fully associative. Furthermore, our F/M-CIP algorithm prevents scan data from being cached at all as opposed to caching them with large re-reference interval value. This is important for SSD cache because of slow writes and wearing issues.

Besides CPU cache replacements, there is an impressive amount of research dedicated to storage cache design such as buffer cache[13, 14, 15, 16, 17, 18, 19, 20, 21, 22] and second level storage cache[23, 24]. While both CPU cache and storage

cache exploit temporal and spatial locality of data accesses, the major difference is that at storage level we usually have more time to allow software implementation of cache algorithms. Therefore, the entire cache is generally fully associative eliminating conflict misses due to set-associative mapping. Cache replacement algorithm becomes the key to cache performance. The two main categories of replacements are recency based and frequency based. LRU has been the dominant cache replacement that performs well for workloads that show mostly recency behavior. To avoid evicting frequently referenced blocks by scans, Least Frequently Used (LFU)[25] algorithm was proposed that can improve the performance of workloads with frequent scans but may degrade performance for workloads that favor recency based cache algorithm. There are studies that combine recency and frequency[25, 13, 14] to provide better performance. ONeil et.al proposed LRU-K[14] which is able to discriminate frequently referenced and infrequently referenced blocks by tracking the times of the last K references to popular database pages. Specifically, the authors discussed LRU-2 for K = 2 which takes the last two references into account. LRU-2 is able to remove cold pages quickly but less effective for workload without strong frequency. LRU-2 uses a priority queue giving rise to logarithm complexity. 2Q[15] introduces one FIFO queue A1in, and two LRU lists: A1out and Am. A missed page is placed in A1in, and its address is stored in A1out if it is replaced. If this page is referenced again and its address is found in A1out, it is promoted to Am where frequently accessed blocks are stored. By tuning the size of A1in and A1out, 2Q can achieve similar performance as LRU-2 with constant overhead.

Instead of looking at recency and frequency of references, Jiang and Zhang proposed LIRS[16] replacement based on Inter-reference Recency. In fact, RRIP[6] share some similar characteristics to LIRS[16] that provides the ability to predict re-reference interval distances based on which replacement is done. LIRS divides

cache into High Inter-Reference Recency (HIR) pages and Low Inter-reference Recency (LIR) pages. LIRS replaces HIR pages and a page in HIR gets promoted to LIR if its IRR is smaller than the maximum recency of all LIR pages. ARC[17] dynamically balances recency and frequency by maintaining two LRU lists, L1 and L2. L1 stores pages that have been seen only once to capture recency, and L2 stores pages that have been seen at least twice to capture frequency. ARC is adaptive because the target size p of T1 which is the top part of L1 is adaptive to workload.

The classic CLOCK algorithm that uses a reference bit and clock hand to approximate LRU algorithm, eliminating lock and list maintenance operations. GCLOCK[20] is a generalized version of CLOCK which associates a counter with each page to carry its weight. CAR[18] maintains two separate clocks and makes CLOCK adaptive to different workloads. CLOCK-Pro[21] uses the reuse distance instead of recency which is similar to LIRS while keeping the overhead low. CLOCK-Pro has shown its effectiveness and been adopted by Linux kernel. The DULO[22] algorithm exploits both temporal and spatial locality of I/O operations. It reorders LRU pages into sequential I/Os in a sequencing bank and then orders the newly formed sequences based on their access recencies in the eviction list to improve disk operation speed without loss of recency. MQ[23] was proposed to improve the performance of second level cache. Multiple and different level LRU queues: Q0, , Qm-1, and a history queue Qout are used to keep warm blocks in high level LRU queues. MQ is effective to catch the frequently accessed blocks that have long distance between two accesses.

While the existing storage cache management algorithms have their respective advantages and disadvantages, none of these algorithms considered the special characteristics of SSD. F/M-CIP algorithm takes into account these physical prop-

erties of SSD and manages the 2 level cache hierarchy in a unique way using the CIP-list. F/M-CIP is able to effectively bypass sweep accesses to large sequential data for which disk performance is acceptable. As a result, it minimizes performance impacts of garbage collection in SSD and avoid write-performance cliff. Furthermore, computation overhead of CIP is a very small.

There has been extensive research reported in the literature in improving SSD cache performance[1, 2, 8, 26, 27, 28] and reducing write wearing[29, 30, 31, 32]. To improve random write performances of SSD, Koltsidas and Viglas have proposed a hybrid SSD and HDD architecture that stores read-intensive pages in SSD and write-intensive pages in HDD[3]. Their online algorithms for optimal page placements have shown great performance advantages over SSD and HDD. Kawaguchi et al[33] developed a UNIX device driver that writes data to the flash memory system sequentially as a Log-structured File System (LFS). The idea has been implemented in todays flash translation layer (FTL) in the controllers of some high end SSDs. Birrell et al[34] presented a technique that significantly improves random write performance of SSD by means of adding sufficient RAM to hold data structures describing a fine grain mapping between disk logical blocks and physical flash addresses. There have been new buffer cache designs that optimize SSD performance by minimizing SSD writes upon cache replacements. Clean First LRU (CFLRU)[35] gives clean pages high priority for evictions until the number of page hits in the working region is preserved in a suitable level. Kim and Ahn proposed Block Padding LRU (BPLRU)[36] buffer cache management algorithm that significantly improves random write performance. While the above studies aimed at optimizing SSD performance considering SSD write problem, F/M-CIP is a unique cache replacement algorithm that dynamically inserts and promotes referenced pages in a specially partitioned LRU list and is able to bypass sequential

accesses without writing to SSD.

SieveStore proposed by Pritchett and Thottethodi[2] elegantly selects data pages to be cached in an ensemble level SSD cache shared by a group of servers. F/M-CIPs candidate-list shares some similar characteristics of selective allocation policy of SieveStore. However, F/M-CIP differs from SieveStore in making dynamic cache management decisions based on online I/O behaviors for the disk cache of each individual server.

3.7 Conclusions

We have presented a new cache replacement algorithm referred to as F/M-CIP that exploits the special physical properties of flash memory and accelerates disk I/O greatly. F/M-CIP divides the traditional LRU list into 4 parts: candidate-list, SSD-list, RAM-list and eviction-buffer-list. Upon a cache miss, the metadata of the missed block is conservatively inserted into the candidate-list but the data itself is not cached. The block in the candidate-list is then conservatively promoted to the RAM-list upon the k-th miss. At the bottom of the RAM-list, the eviction-buffer accumulates LRU blocks to be written into the SSD cache in batches to exploit the internal parallelism of SSD. The SSD-list is managed using a combination of recency and frequency replacement policies by means of conservative promotion upon hits. A prototype F/M-CIP driver has been built on Linux kernel at the generic block layer. Experimental results on standard benchmarks and real world traces have shown that F/M-CIP speedups disk I/O performance by an order of magnitude compared to traditional hard disk storage and up to a factor of 3 compared to the traditional SSD cache algorithm in terms of application execution time. Experimental results have also shown that F/M-CIP substantially reduced write operations giving rise to longer life time of flash memory SSD.

Acknowledgment

This research was sponsored in part by the National Science Foundation under grants CCF-1421823, CCF-1439011, and CCF-1017177. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

List of References

[1] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 327–338.

[2] T. Pritchett and M. Thottethodi, "Sievestore: a highly-selective, ensemble-level disk cache for cost-performance," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 163–174.

[3] I. Koltsidas and S. D. Viglas, "Flashing Up the Storage Layer," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 514–525, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453913

[4] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250709

[5] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[6] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 60–71. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815971

[7] H. Gao and C. Wilkerson, "A dueling segmented LRU replacement algorithm with adaptive bypassing," in *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, 2010.

[8] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 278–289.

[9] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 266–277.

[10] M. Srinivasan, P. Saab, and V. Tkachenko, "Flashcache," https://wiki.archlinux.org/index.php/Flashcache.

[11] D. A. Lesmond, J. P. Ogden, and C. A. Trzcinka, "A new estimate of transaction costs," *Review of Financial Studies*, vol. 12, no. 5, pp. 1113–1141, 1999.

[12] U. of Massachusetts, "SPC Traces," http://traces.cs.umass.edu/index.php.

[13] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-based Replacement," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '90. New York, NY, USA: ACM, 1990, pp. 134–142. [Online]. Available: http://doi.acm.org/10.1145/98457.98523

[14] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93. New York, NY, USA: ACM, 1993, pp. 297–306. [Online]. Available: http://doi.acm.org/10.1145/170035.170081

[15] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672996

[16] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '02. New York, NY, USA: ACM, 2002, pp. 31–42. [Online]. Available: http://doi.acm.org/10.1145/511334.511340

[17] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of FAST'03*, vol. 3, 2003, pp. 115–130.

[18] S. Bansal and D. S. Modha, "CAR: Clock with Adaptive Replacement," in *Proceedings of FAST'04*, vol. 4, 2004, pp. 187–200.

[19] F. J. Corbato, "A paging experiment with the multics system," DTIC Document, Tech. Rep., 1968.

[20] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing," in *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '92/PERFORMANCE '92. New York, NY, USA: ACM, 1992, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/133057.133084

[21] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 323–336.

[22] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proceedings of FAST'05*, vol. 4. San Francisco, CA, USA, 2005, pp. 8–8.

[23] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 6, pp. 505–519, June 2004.

[24] T. M. Wong and J. Wilkes, "My Cache Or Yours?: Making Storage More Exclusive," in *USENIX Annual Technical Conference*, 2002, pp. 161–175.

[25] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," *Computers, IEEE Transactions on*, vol. 50, no. 12, pp. 1352–1361, Dec 2001.

[26] L.-P. Chang and T.-W. Kuo, "Efficient Management for Large-scale Flash-memory Storage Systems with Resource Conservation," *Trans. Storage*, vol. 1, no. 4, pp. 381–418, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1111609.1111610

[27] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508270

[28] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*,

ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[29] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669118

[30] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, ser. Proceedings of FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855518

[31] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-based Write Caches," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, ser. Proceedings of FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855519

[32] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-based SSDs," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960482

[33] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in *Proceedings of USENIX Technical Conference*, 1995, pp. 155–164.

[34] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-performance Flash Disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243418.1243429

[35] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 234–241. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176789

[36] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of FAST'08*, vol. 8, 2008, pp. 1–14.

MANUSCRIPT 4

A New Metadata Update Method for Fast Recovery of SSD Cache

Jing Yang[1], Qing Yang[2]

is published on the 8th IEEE International Conference on Networking, Architecture and Storage.

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: jyang@ele.uri.edu

[2]Distinguish Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: qyang@ele.uri.edu

Abstract

In order to maintain cache data durable after a crash or reboot, metadata information needs to be stored in a persistent storage. There are several ways to update metadata. Update-write-update and write-update are the most typical ones. While write-update method reduces one metadata update write, it limits the amount of data in cache that can be used after a system crash. We present a design and implementation of a novel metadata update method for SSD (solid-state disk) cache of data storage systems, referred to as Lazy-Update Following a Update-Write (LUFUW). Our new metadata update method allows maximal amount of data in SSD cache available upon restart after a power failure or system crash with minimal additional writes to SSD. This capability makes restart run twice as fast as existing SSD caches such as Flashcache [1] that can only use dirty data in the cache after crash recovery. We present our prototype implementation on Linux kernel and performance measurements as compared with existing SSD cache solutions.

4.1 Introduction

Recent developments of flash memory based SSD (solid state disk) provides us with great advantages in terms of high storage performance, low-energy, compact size, and shock resistance. The current price gap and speed difference between hard disk and SSD make SSD a perfect choice as a cache layer between system RAM and disk storage [2, 3, 4, 5, 6]. The nonvolatile characteristic of SSD allows the cached data persistent even after power failures or system crashes so that the system can benefit from hot restart. Taking a 500 IOPS disk with 100GB cache system as an example, it will take over 14 hours to fill a cold cache after a system reboot or crash [7]. Therefore, a hot SSD cache after a restart makes a significant difference in storage performance. However, current researches on SSD caches have mainly focused on cache architecture or management algorithms to optimize performance under normal working conditions. Little study is done on exploiting SSD caches durability across system crashes or power failures.

Hot restart can be achieved by saving parts of caches mapping and state information (referred to as metadata) in SSD. When system is normally shut down, those metadata can be consistently written back to SSD. However, metadata update is needed to keep cache consistent for recovery purpose after unexpected system crashes or power failures, which is more expensive. There are two ways to update metadata, either using log to record metadata change, or update in-SSD metadata directly. Some SSD cache, such as Flashcache, updates in-SSD metadata synchronously when data is changed to dirty state. Traditionally, an update-write-update operation is needed for each dirty write. The first update indicates which block to be written. The following update confirms the write after the data is successfully written. As a result, one data write requires three SSD write operations. In order to reduce metadata writes to SSD, Flashcache uses write-update instead,

i.e. metadata update follows a successful data write. In this case, if there is a crash after a data write but before its metadata update, cache manager does not know which block has been written, giving rise to the possibility that non dirty block may be in an inconsistent state. To guarantee correctness upon recovery, only dirty blocks in the SSD cache can be used. Our recovery experiments have shown that, with this metadata update method, only a small portion of cached data can be used after restart as shown in Figure 22 .

To enhance performance with a hot restart, a new metadata update technique is proposed in this paper to maximize useable data upon a system restart in events of crash or power failure. The idea is to write metadata to SSD before writing data for each SSD write operation. Each metadata entry keeps a flag indicating the corresponding data is in the process of being written to SSD. As soon as the data is successfully written to SSD, this flag is reset in the RAM copy of the metadata. When next time this metadata with the flag reset is written to SSD, it becomes consistent with the data in the SSD cache. Since metadata in SSD is a simple address map, each 4KB page, the basic write unit in SSD, contains over a hundred of metadata entries. Every time there is a metadata update, many such flags are reset in SSD fairly frequently as a free ride. As a result, only a very

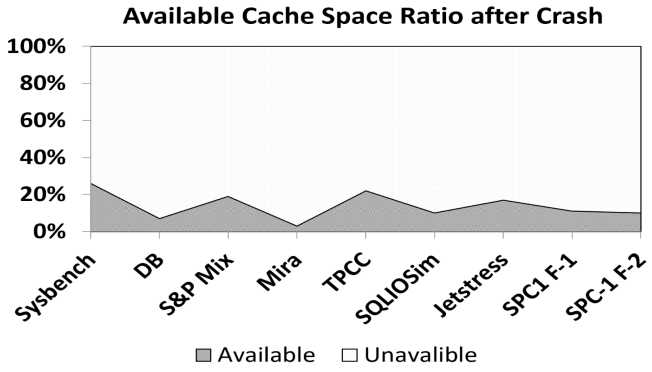


Figure 22: Available cached data upon restart after a crash using write-update metadata method.

small portion of cached data in the SSD have this flag set indicating inconsistency and therefore unusable after restart. Since the confirmation updates of metadata are done lazily, we name our metadata update method Lazy-Update Following Update-Write, LUFUW for short. A sliding window and a time stamp are also used to keep track of the unfinished writes. This sliding window and time stamp are written to SSD together whenever a metadata page is written to SSD. In this way, we can further confirm the writes in an older set which does not appear in a newer sets sliding window. A forced flag reset is performed when the window size reaches a predetermined maximal length, further reducing amount of inconsistent data in cache upon restart.

To quantitatively evaluate the performance of our new metadata update method, a prototype has been built on top of Flashcache which appears as a device driver at the generic block layer in the Linux kernel. Standard benchmarks and I/O traces were used to drive the prototype measuring I/O performance in comparison with Flashcaches original metadata management method. Experimental results on standard benchmarks and traces have shown that our new metadata update method allows over 99% of cached data usable at restart as compared to less than 14% with Flashcaches original metadata update method. Because of lazy updates for confirmations that are mostly free ride as piggybacks of other updates, additional metadata overhead is minimal. The new metadata update method makes restart run twice as fast as Flashcache that can only use dirty data in the cache after crash recovery.

In summary, the major contributions of this paper are:

1. A novel metadata update technique has been presented for durable SSD cache. The new technique does not increase metadata update overhead but ensures a hot restart of the SSD cache after a power failure and system crash.

2. A working prototype has been built and tested in Linux Kernel.

The paper is organized as follows. Section 4.2 gives the design and implementation of the prototype. Experimental settings and workload characteristics are presented in Section 4.3 followed by results and discussions in Section 4.4. We discuss related work in Section 4.5 and conclude the paper in Section 4.6.

4.2 Prototype Design and Implementation

For a graceful shutdown or restart, metadata in memory can be written into SSD in a consistent state after getting a notification from the system. But for an unexpected system crash or power failure, the metadata may be in an inconsistent state. Therefore, it is necessary to do real time metadata update in SSD for write I/Os. A transaction like update-write-update is the typical and safest way to update metadata. The first metadata update indicates which entry will be written. Then, after the data is successfully written in SSD, the later update confirms metadata complete. If a system crash happens, it is easy to find which entry in the cache is in an inconsistency state. While write-update has been implemented to reduce the two additional metadata update writes to just one, there is a potential danger of data corruption. The reason is, if a crash happens after a data is written but before metadata is updated in SSD, the cache entry contains the data inconsistent with its corresponding metadata (address map and dirty/clean state). That is, the data in that location has been changed but the change has not been reflected in the SSD metadata. To avoid such data inconsistency and guarantee correctness when system restarts, only the blocks that are in dirty state can be used after system restart. No clean blocks can be used because a clean block is not guaranteed to be the same as the one on disk.

In order to maximize consistent data blocks in the cache with minimal overhead of metadata update, our solution is using lazy update. In our method, we

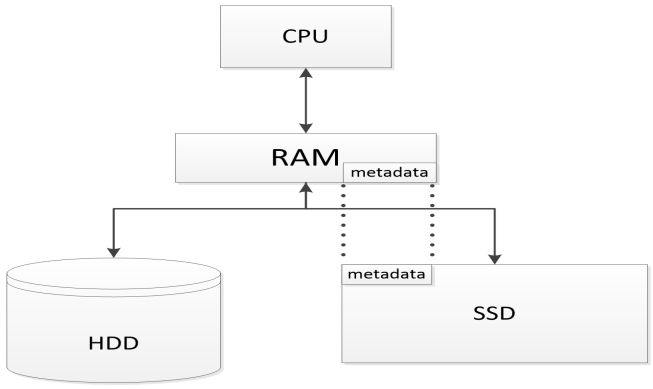


Figure 23: System architecture for LUFUW cache.

update metadata first followed by data write. The confirmation metadata update is done lazily as a free ride of later metadata update in the same metadata block. The overall architecture is shown in Figure 23 with metadata reside in both DRAM and SSD. Figure 24 shows the process of a write in our new LUFUW system. Upon a write I/O, the in-RAM metadata will be updated first to show which entry in the SSD cache will be written. Then the block containing the updated metadata will be writing to SSD as a metadata update write. When the data write finishes, the in memory metadata will be updated again before the write returns to the system. Later, the confirmation update will be reflected to SSD as a free ride when this block of metadata needs to be written to SSD as the first metadata update for other write I/Os.

For each metadata entry, in addition to address map and cache states LUFUW maintains one additional 1-bit flag indicating that the write is in progress. We call it SSD-writing bit. The flag is set to 1 initially when metadata is written to the SSD. As soon as the data is successfully written in SSD, we reset this flag to 0 in the RAM copy of the metadata only. This flag reset will eventually go to SSD through a free ride with other metadata update in the same metadata page, referred to as lazy-update. The metadata state transition is shown in Figure 25. In our design each metadata page contains 240 metadata entries. The chance of

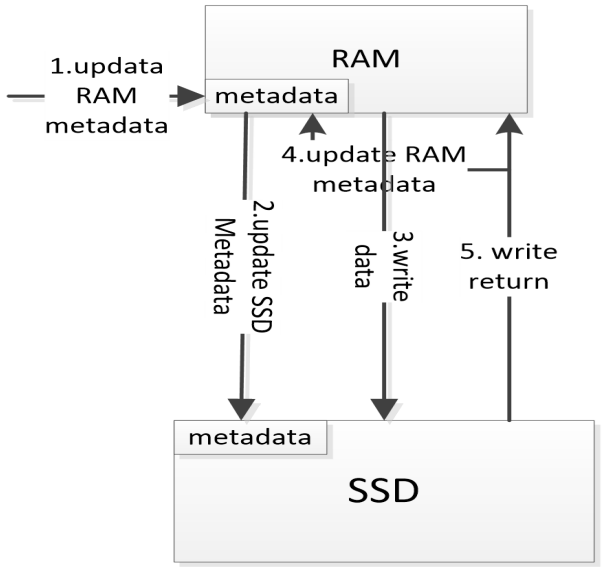


Figure 24: Write process for LUFUW.

a flag being frequently updated in SSD is very high because of data locality. In the worst case, each metadata page will contain at least one unconfirmed. Taking a 320GB SSD cache as an example, there may be over 1 gigabyte space that could not be used after crash. To further increase the valid cache data range after recovery, a sliding window and a time stamp are used to keep track of the unfinished writes. This sliding window and time stamp are written to SSD together whenever a metadata page is written to SSD. The sliding window contains unfinished data writes. When a write comes, the index of the writes metadata block will be inserted into the window. The index will be removed after the write returns. To enhance time consistency, the window is implemented in a similar way as TCP/IPs slide window in network design. The window will not slide without the oldest one being confirmed. In this way, we can further confirm the writes in an older metadata page which does not appear in a newer metadata pages slide window. In our current design, the maximal window length is 63 entries and this window together with a time stamp is piggybacked to every metadata page written to SSD. When system restart after a crash, we only need to find the sliding window with the latest time

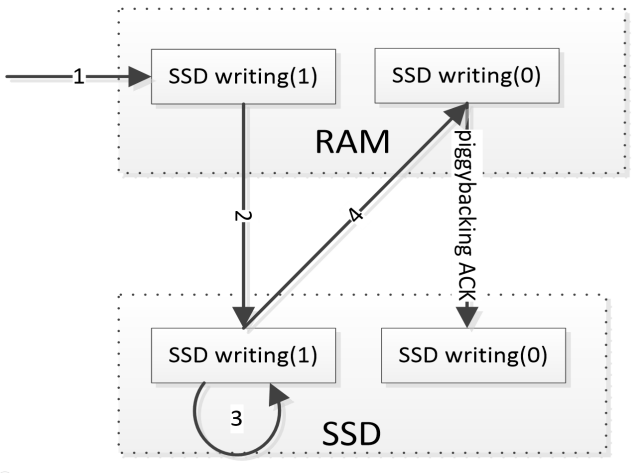


Figure 25: Metadata state transition for LUFUW.

stamp and make sure their corresponding metadata pages unconfirmed entries and data are not usable. All other data blocks are valid and can serve I/Os. Note that a piggybacked window may be smaller than 63 entries. To prove the concept of our design, we have developed a working prototype based on Flashcache under the Linux kernel.

We implement our new LUFUW metadata policy in Flashcahe. Flashcache is based on Linux Device Mapper (DM). Unlike LVM2 EVMS and software RAID that combine the address space of all disks, Flashcache map SSD to hard disk and set the address space of SSD to 0. In this way, Flashcache can manage SSD as a cache. It is implemented as a device driver, so it can be easily used in many Linux applications including database system and other applications [8, 9, 10]. After Flashcache driver is inserted into Linux kernel, it appears as a device. So Flashcache only processes block I/Os sent to it, and does not affect other system parts. In order to implement our LUFUW metadata update method, we need not only implement our method in Flashcaches write routine, but also in the routine of read misses in Flashcache. Since Flashcache only reuse dirty blocks in the cache after a system crash or power failure, it does not update metadata in the read

miss path because they are not dirty data in SSD cache. In order to make use of maximal amount of cached data upon restart, LUFUW also updates metadata on read misses in SSD.

4.3 Experimental Setup and Workload Characteristics

4.3.1 Experimental Setup

In order to quantitatively evaluate and compare the recovery performance and the overhead of LUFUW with existing cache metadata update policies, we have installed the LUFUW prototype on 3 Dells PowerEdge R310 rack servers running Linux in our lab. Each rack server has an Intel Xeon X3460 processor with 2.80 GHz and 8M Cache. The system RAM of each server is 8GB with 1333MHz and Dual Ranked UDIMM. If system RAM is larger than working data size, most I/O will be cached by system RAM cache. So we restricted RAM used by Linux through GRUB command in different experiments. The hard disk drive used in our tests is 1TB 7.2K RPM SATA drive. We have also installed the original Facebooks Flashcache. We used Flashcaches version 1.0.121. We carried out our experiments on both native and virtualized environment using KVM.

4.3.2 Workload Characteristics

The workloads we used in our experiments consist of two standard benchmarks and one real world I/O traces. The standard OLTP benchmark, SysBench [11], is a modular, cross-platform, and multi-threaded benchmark for systems under intensive I/O loads. We used SysBench in two ways. First we initiated 100 million rows Database (DB) in physical machine server and then tested the DB with 100,000 transactions through 16 threads.

Server consolidation and cloud computing have recently become very popular through virtualizations. In such environment, multiple virtual machines that carry out different applications are installed and run on a single physical server host. To

evaluate how such mixed workloads affect the performance of LUFUW cache, we have set up multiple virtual machines (VMs) that ran different benchmarks on one host machine. In such a system, multiple virtual machines running benchmarks generate I/Os to the host hypervisor where our storage drivers are installed. SysBench and PostMark [12] were chosen as typical workloads for their different I/O characteristics. SysBench has high locality while PostMark does not. The storage system receives aggregated I/O requests from all virtual machines. This is an interesting performance test for various storage architectures and is close to real world applications.

A real world trace [13] is also used in the tests. The trace is from a small Microsoft data center. It was collected from 36 different volumes on 13 servers in a period of 7 days. We simulate the data center using six volumes in six virtual machines running concurrently. The six virtual machines all run above the cache so that all virtual machines I/Os will go through the cache.

4.4 Results and Discussions

In this section, we report and discuss our experimental results. To have a fair performance comparison, we run Flashcache with and without the new metadata update mechanism.

Using SysBench as benchmark, we first run 50,000 transactions to the database for warming up the cache. Then we reboot gracefully for normal recovery. We also manually unplugged the power of the machine and restart for crash recovery. After restart, 5,000 transactions are sent to the database to measure the performance. Figure 26 shows the measured results in terms of execution time after system restarts. We used empty SSD cache or cold cache as the baseline for comparison (BL) which is the first bar in the figure. In case of graceful reboot, the entire metadata is updated in the SSD upon receiving a shutdown signal. Therefore,

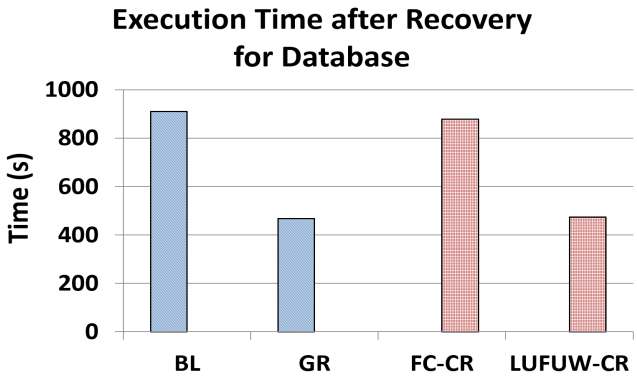


Figure 26: Restart performances with different metadata update schemes for database benchmark.
• BL: baseline, system reboot from an empty cache.
• GR: graceful reboot, metadata updated in SSD upon shutdown signal.
• FC-CR: Flashcaches crash recovery.
• LUFUW-CR: Crash recovery of LUFUW metadata update mechanism.

restart performance would be ideal since the cache is warm and the entire cache is useable upon restart. The execution time of graceful reboot is shown as the second bar in Figure 26.

The restart performance of native Flashcache software is shown as the third bar (FC-CR) of Figure 26. Because only the dirty blocks in the SSD cache are useable upon restart after a crash, the restart performance is not much different from cold restart baseline (BL) implying that the cache is pretty cold upon restart. Because of the fact that majority of cached data (clean data) are not useable after restart, we observed only 4% difference between BL and FC-CR. The forth bar in Figure 26 shows the execution time of LUFUW method. We observed substantial performance improvement over traditional metadata update method. The improvement is 92% over cold cache BL. This significant improvement can be attributed to the large amount of useable data in the SSD cache upon restart due to the efficient metadata management technique.

In addition to the standard benchmark, Microsoft small data center traces are used to measure our new metadata update method. We first run 1,200,000 I/Os

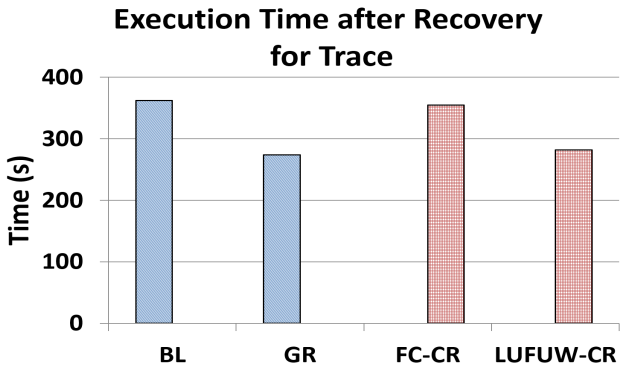


Figure 27: Restart performances with different metadata update schemes for I/O traces.
• BL: baseline, system reboot from an empty cache.
• GR: graceful reboot, metadata updated in SSD upon shutdown signal.
• FC-CR: Flashcaches crash recovery.
• LUFUW-CR: Crash recovery of LUFUW metadata update mechanism.

(200,000 I/Os per virtual machine) to warm up the cache. After this, we reboot or power off the machine in the same way as we did in the database recovery experiment. Then we send another 360,000 I/Os to measure the performance. The result is shown in Figure 27. While the trend in Figure 27 is similar to that of Figure 26, the difference between the new metadata update method and the traditional one is not as dramatic. Our new method is about 20.5% faster than Flashcaches crash recovery and the difference between Flashcaches crash recovery and baseline is about 2%. This is mainly because the trace tests are done in a virtual environment that needs virtual machines (VM) to run. Upon restart after recovery, all VMs where the traces are replayed need to be restarted first. Since the virtual machines are also above the cache, restarting the virtual machines will cause some hot data to be flushed out of the cache reducing the cache hit ratio of restart process. But we still observe significant improvement of our new method over traditional metadata update method.

The next question to be asked is what the cost of such high restart performance would be. We measured the additional writes to SSD as results of metadata

updates for both Flashcache and the new LUFUW as shown in Figure 28. As mentioned in Section 4.2, the number of SSD metadata update writes should increase because LUFUW also updates metadata in the read miss routine. While the result of SysBench shows LUFUW has 12.3% more metadata writes than Flashcache, the mixed workloads of SysBench and PostMark and Microsoft data center traces show just the opposite. In the mixed workloads of SysBench and PostMark, LUFUW has 22.8% less metadata writes than Flashcache. Microsoft data center I/O traces show 48.9% less metadata writes than Flashcache. Let us take a closer look at the results of the mix benchmark to understand why this is happening. While the hit ratios of the two metadata update methods are very close, LUFUW batches 73.8% more metadata update writes than Flashcache resulting in much less total metadata writes to SSD. The reason why LUFUW can batch so much more metadata updates in each metadata page write is two folds. First of all, LUFUW writes metadata to SSD first before data is written, whereas Flashcache write metadata to SSD after data is successfully written to SSD. Therefore, LUFUW collects and batch metadata updates from the I/O queue to the SSD device while Flashcache collects and batch metadata update from returned write I/Os from the SSD device. Because of data locality and high speed CPU, the chance of finding write I/Os that have metadata sharing a same metadata page in the I/O queue is much higher than among returned I/Os that are typically separated by large time interval. The second reason is that we observed a fairly good chance that many overwrites to the same location during the time a metadata page is being updated in the SSD. LUFUW can take full advantage of this write locality while original Flashcache cannot. This is particularly true for Microsoft data center I/O traces due to its high access locality [3]. The time interval between two I/Os in the same LBA is very small for the hot blocks. LUFUW updates metadata first and

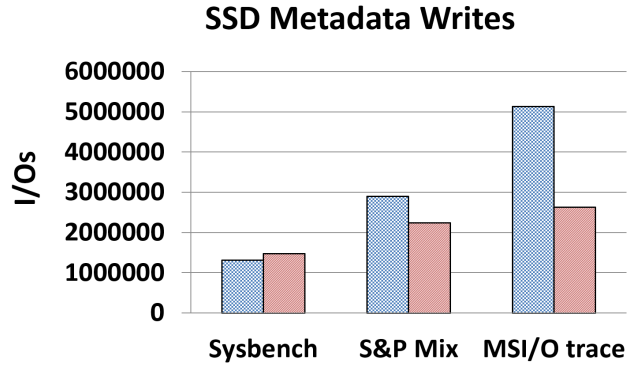


Figure 28: Metadata update overhead.

sets the dirty bit of a write earlier than Flashcache does. This can increase the dirty hit chance of the following write in the same location eliminating unnecessary additional metadata updates and reducing the total the number of metadata updates.

In term of cache performance for normal operations, the two metadata update methods are very close as shown in Figures 29 and 30. In particular, the performance of LUFUW in the mixed benchmark is slightly better than Flashcache. This is because the proportion of metadata writes is over 15% of the total SSD writes. For the other two benchmarks, metadata writes constitute less than 4% total write I/Os. Our new metadata update method needs more process time resulting in increased the response time. However, the difference is negligible as shown in the figures. The benefits of fast recovery and restart are substantial as demonstrated in our experiments.

4.5 Related Work

Flash memory SSD has emerged as a promising storage media and fits naturally as cache between system RAM and disk due to its performance/cost characteristics. With fast advances of flash memory technologies, there has been extensive research reported in the literature in improving SSD cache perfor-

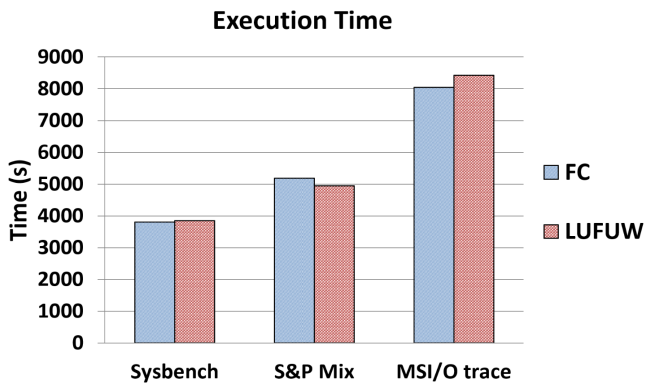


Figure 29: Execution time for three benchmarks.

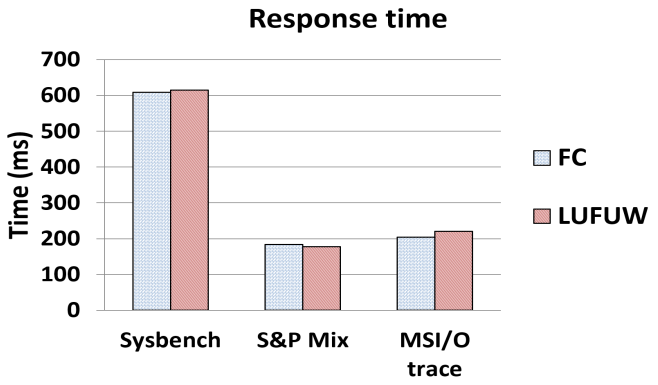


Figure 30: Response time for three benchmarks.

mance [3, 6, 14, 15, 16, 17] and reducing write wearing [15, 18, 19, 20]. To improve random write performances of SSD, Koltsidas and Viglas have proposed a hybrid SSD and HDD architecture that stores read-intensive pages in SSD and write-intensive pages in HDD [4]. Their online algorithms for optimal page placements have shown great performance advantages over SSD and HDD. Kawaguchi et al. [21] developed a UNIX device driver that writes data to the flash memory system sequentially as a Log-structured File System (LFS). The idea has been implemented in todays flash translation layer (FTL) in the controllers of some high end SSDs [22]. Birrell et al. [23] presented a technique that significantly improves random write performance of SSD by means of adding sufficient RAM to hold data structures describing a fine grain mapping between disk logical blocks and

physical flash addresses. There have been new buffer cache designs that optimize SSD performance by minimizing SSD writes upon cache replacements. Clean First LRU(CFLRU) [24] gives clean pages high priority for evictions until the number of page hits in the working region is preserved in a suitable level. Kim and Ahn proposed Block Padding LRU (BPLRU) [25] buffer cache management algorithm that significantly improves random write performance. A recently proposed design called SieveStore [3] elegantly selects popular blocks to store in SSD cache. Two variants of SieveStore, SieveStore-C and SieveStore-D have been provided in their paper. Sieve Store-C uses two levels counters to sieve hot blocks while SieveStore-D keeps the precise access counts of blocks in one level. Canim et al. prototyped a SSD cache as an extension of the in-memory buffer pool named Temperature-Aware Caching (TAC) [9]. Aiming at storing more hot data in SSD, TAC maintains temperature information of disk data. It divides disk into regions, and maintains temperature information in every region as disk accesses occur. Do et al. [26] extends the work of Canim et al. They implemented three SSD design alternatives. But different from TAC, page reads from disk (clean page) are delayed to write to SSD. The write will happen only after the page is evicted from buffer cache. This will reduce writes to SSD and improve performance.

While the above studies aimed at optimizing SSD performance, there is only a few exploiting on SSD caches nonvolatile feature.

In order to maintain cache data durable after crash or reboot, metadata needs to be stored in persistent storage. In-SSD metadata is persistent and can keep the cache consistent. Typically, there are two ways to update metadata in SSD. One is using log to record metadata changes and the other is updating in-SSD metadata directly upon write I/Os. The first one is known as logging which is relatively simple. Every log entry may contain cache blocks state and the mapping table.

Sometimes it also contains a sequence number. Log size increases as write I/Os are issued. In order to restrict the log size, a checkpoint can be placed [7]. Maintaining a log requires cleaning and garbage collection that demand more RAM and CPU resources and may negatively impact system performance. FaCE [10] is an SSD cache using the log like metadata update method. FaCE maintains its metadata similar to the databases log system. Its metadata changes are written to SSD in a single large segment. A checkpoint is used to determine if the segment is in a consistent state.

Direct in-SSD metadata update generally allocates a small part of SSD to store metadata. Metadata is updated directly upon a write. Flashcache [1] is the representative of direct in-SSD metadata update. It updates metadata after data has been written to SSD. Metadata stored in SSD are only updated after a SSD block status changes from clean to dirty or from dirty to clean. Other state changes will not cause data inconsistency. In normal restart case, Flashcache reads the metadata table in SSD and uses all the entries in it to rebuild the cache. In case of crash, only dirty entries in the table are used to rebuild the cache. Flashcache also uses checksums to handle bit corruption and short writes caused by events such as power failures [27] and internal errors. Another work is done by Bhattacharjee et al. [8]. They have extended their work in TAC to map the slot directory (metadata) into a flash resident file. As a result, the metadata changes become file changes and are reflected to the file in the cache.

4.6 Conclusions

We have presented a new SSD cache metadata update algorithm referred to as LUFUW to make SSD cache durable. Our objective is to maximize cache performance upon a system restart after a crash, power failure, or reboot. LUFUW allows majority of cached data useable when system restarts by means of the new

metadata update technique that exploits data locality. LUFUW does metadata update first before data writes. The confirmation update that is essential to guarantee data consistency is done lazily as a free ride with other metadata updates. Such lazy updates substantially reduce SSD write operations whereas keeping the inconsistency window minimal (less than 1% of cached data). A working prototype has been implemented in Flashcache which is a device driver in Linux kernel. Experimental results on standard benchmarks and real world traces have shown that the new metadata update method makes restart run twice as fast as original Flashcache design. Furthermore, metadata processing overhead is kept very low.

List of References

[1] Facebook, "Flashcache," https://github.com/facebook/flashcache.

[2] C. Dirik and B. Jacob, "The Performance of PC Solid-state Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 279–289. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555790

[3] T. Pritchett and M. Thottethodi, "Sievestore: a highly-selective, ensemble-level disk cache for cost-performance," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 163–174.

[4] I. Koltsidas and S. D. Viglas, "Flashing Up the Storage Layer," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 514–525, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453913

[5] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2011, pp. 1–11.

[6] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 327–338.

[7] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 267–280.

[8] B. Bhattacharjee, K. A. Ross, C. Lang, G. A. Mihaila, and M. Banikazemi, "Enhancing recovery using an ssd buffer pool extension," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM, 2011, pp. 10–16.

[9] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "Ssd bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[10] W.-H. Kang, S.-W. Lee, and B. Moon, "Flash-based extended cache for higher throughput and faster recovery," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1615–1626, 2012.

[11] "SysBench," http://sysbench.sourceforge.net/.

[12] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, Tech. Rep., 1997.

[13] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 10, 2008.

[14] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 278–289.

[15] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669118

[16] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[17] L.-P. Chang and T.-W. Kuo, "Efficient Management for Large-scale Flash-memory Storage Systems with Resource Conservation," *Trans. Storage*, vol. 1, no. 4, pp. 381–418, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1111609.1111610

[18] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *Proceedings of the 8th USENIX*

Conference on File and Storage Technologies, ser. Proceedings of FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855518

[19] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-based Write Caches," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, ser. Proceedings of FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855519

[20] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-based SSDs," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960482

[21] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in *Proceedings of USENIX Technical Conference*, 1995, pp. 155–164.

[22] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *Consumer Electronics, IEEE Transactions on*, vol. 48, no. 2, pp. 366–375, May 2002.

[23] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-performance Flash Disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243418.1243429

[24] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 234–241. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176789

[25] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of FAST'08*, vol. 8, 2008, pp. 1–14.

[26] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging dbms buffer pool using ssds," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 1113–1124.

[27] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of ssds under power fault," in *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, 2013, pp. 271–284.